Enhancing Neural Network Robustness via Synthesis of Repair Programs

Tom Yuviler[0009-0008-7952-8292] and Dana Drachsler-Cohen[0000-0001-6644-5377]

Technion, Haifa, Israel `{tom.yuviler@campus,ddana@ee}.technion.ac.il`

Abstract. Adversarial examples undermine the reliability of neural networks. To defend against attacks, multiple approaches have been proposed. However, many of them introduce high training overhead or high inference overhead, some significantly decrease the network's accuracy or insufficiently increase the network's robustness, and others do not scale to deep networks. To mitigate all these shortcomings, we propose a new form of defense: optimal program synthesis of short *repair programs*, integrated into a trained network. A repair program modifies a few neurons by using a few other neurons. The challenge is to identify the most successful combination of neurons to enhance the network's robustness while maintaining high accuracy. We introduce `DefEnSyn`, a stochastic synthesizer of repair programs. To cope with the exponential number of neuron combinations, `DefEnSyn` learns the effective combinations by synthesizing repair programs of increasing length. We evaluate `DefEnSyn` on classifiers for ImageNet and CIFAR-10 and show it enhances the robustness of networks to L_∞-, L_2-, and L_0- black-box adversarial example attacks and to backdoor attacks. `DefEnSyn`'s repair programs enhance the networks' robustness on average by +40% and up to +71%. `DefEnSyn` decreases the network's accuracy by only $\approx -1\%$. We demonstrate that `DefEnSyn` outperforms existing state-of-the-art defenses based on adversarial training, randomization, and repair, in both robustness and accuracy.

Keywords: Neural Network Robustness · Program Synthesis.

1 Introduction

Despite the immense success of neural networks, their reliability is still an open challenge due to their vulnerability to various kinds of attacks. One of the widely studied attacks is the adversarial example attack, where an adversary computes a small perturbation that causes the network to predict the wrong output [26,36,73,4,84,50,11,71]. When the attacked network is an image classifier, the attacker typically aims at generating an imperceptible perturbation. Formally, the attacker computes a perturbation whose magnitude is smaller than a predefined small threshold, where the magnitude is measured with respect to a given p-norm, such as L_∞ [26,50], L_2 [11,4], L_1 [17,12], or L_0 [71,16]. These

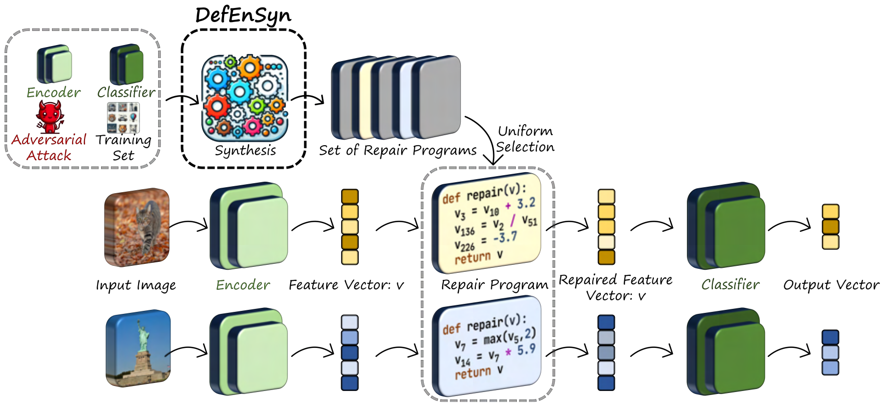
Table 1. Adversarial defenses: advantages and disadvantages (yes ✓, partly ✓, no ×).

	High clean accuracy	High robust accuracy	Scalability for large networks	Low training overhead	Low inference overhead
Adv. training	✓	✓	✓	×	✓
Randomization	✓	✓	✓	✓	✓
Repair	✓	✓	✓	✓	✓
`DefEnSyn` (ours)	✓	✓	✓	✓	✓

adversarial examples raise concerns about deploying AI in safety-critical applications, leading to new regulations by the European Union [22] and guidelines for secure AI system development by the NSA [54].

To mitigate adversarial attacks, many *adversarial defenses* have been proposed. An adversarial defense aims to make a network more robust to adversarial attacks without significantly decreasing the network's accuracy. This is commonly obtained by modifying the network's computations or the input to the network during training or inference. Existing adversarial defenses are thus required to carefully balance all these conflicting goals: high accuracy on unperturbed inputs (called *clean accuracy*), high accuracy on adversarially perturbed inputs (called *robust accuracy*), low training overhead, low inference overhead, and scaling to large networks. This has led to three kinds of defenses: adversarial training, randomization, and post-training repair (repair for short). Adversarial training alters the training to consider adversarial examples [66,64,79,50,36,1]. Often, these defenses obtain high clean accuracy and high robust accuracy (with respect to the given attack types). However, they introduce significant overhead in computational resources and training time (e.g., several days). This overhead becomes significant when newly discovered attacks necessitate repeated retraining of the network to enhance its robustness. Randomization adds stochastic noise to the input [62,82,9,15,10,53,39] or to the network's computations [45,37]. While randomization defenses have a lighter training overhead than adversarial training, they tend to decrease the network's clean accuracy and still introduce some time overhead during both training and inference. Repair techniques introduce post-training modifications to the neural network, like parameter adjustments [77,74,72,67,49] or architectural modifications [38,70,23]. Most repair techniques do not aim to defend against adversarial examples but rather repair benign misclassifications [49,74,67,70,23] or enforce specifications, e.g., linear constraints over the network's output [38]. However, repair has also been proposed for adversarial defense [77]. Its main disadvantage is that it often relies on expensive analysis, which does not scale to large networks. There are two exceptions [74,70], but they focus on provable repair, which becomes infeasible when the number of inputs required to be defended is more than a few hundred. Table 1 summarizes the advantages and disadvantages of the defense types.

We propose a novel form of defense: optimal program synthesis [51,13,8] of *short repair programs*, integrated into a trained network, that enhance robust-



```
def repair(v):
  v_3 = v_10 + 3.2
  v_136 = v_2 / v_51
  v_226 = -3.7
  return v
```

```
def repair(v):
  v_7 = max(v_5,2)
  v_14 = v_7 * 5.9
  return v
```

Fig. 1. Illustration of `DefEnSyn`, an adversarial defense via optimal program synthesis.

ness with a minimal decrease in the clean accuracy. Program synthesis relies on training, enabling us to look for a repair that maximizes both the robust and clean accuracies. It is also commonly restricted to a short solution, enabling us to lower the training and inference overhead. To scale to deep networks, a *repair program* is integrated into a trained network between the network's *encoder* and *classifier* (illustrated by Figure 1). The encoder consists of the first layers, transforming an input into a lower-dimensional feature vector, and the classifier consists of the other layers, mapping the feature vector to a probability vector over the classes. Defending a network by modifying the network's extracted features has been shown to be successful [83,41]. Our repair programs modify the values of a *few* neurons using a *few* other neurons. This idea builds on the observation that certain parts of neural networks act as *program modules* and can be recomposed to achieve a task *without retraining* [55,58]. Our defense has several advantages: (1) it is applicable to any network architecture and does not make any assumptions about its layer types, (2) it is computed *after* the network has been trained, and (3) during synthesis, evaluating candidate programs requires only the classifier part, keeping the synthesis overhead relatively low, even for deep networks. Our focus is *optimal* program synthesis since the goal of adversarial defense is to *maximize* the clean and robust accuracies on a given training set. This is different from provable repair [74,70], which looks for a repair with perfect accuracy on the training set and fails if there is none. Naturally, for large training sets (more than a few hundred), provable repair fails more often.

Computing a repair program that maximizes the clean and robust accuracies is highly challenging. First, it requires identifying the best neurons to include in the repair program (on the left-hand side and the right-hand side of its instructions) out of an exponential number of possibilities. The challenge is that neurons are not standard program variables but rather functions over the network's inputs, and some of them are correlated. Second, there is no monotonicity between the effectiveness of a program and programs that extend it. Thus, greedy synthesis, iteratively generating the next best instruction, is unsuitable. Third, stochastic synthesis that samples full programs (e.g., [60,21]) requires an infeasible number of samples to converge. To illustrate, if there are 100 neurons and the repair program can have up to ten instructions, only checking a single program for each possibility of repaired neurons requires 10^{13} samples. Even if evaluating a single sample takes one millisecond, this basic sampling requires over 300 years. To the best of our knowledge, choosing a small set of variables out of a large set of correlated variables is a new challenge in program synthesis.

We introduce `DefEnSyn`, a synthesizer that computes a set of repair programs. It relies on two ideas. First, to cope with the exponential number of possibilities, it learns the effectiveness of each neuron *separately* (not as part of sets). Second, although there is no monotonicity between programs and their extensions, it uses short programs, whose search space is smaller, as guidance towards the effective neurons in longer programs. Technically, it learns two distributions over the neurons, for the left-hand side of the instructions (the *repaired* neurons), and for the right-hand side (the *repairing* neurons). To this end, it iterates over program lengths from 1 to k. For each, it samples full programs based on the distributions learned by previous iterations. It updates these distributions by the average accuracies of candidate programs. This sampling method helps it to identify the more suitable neurons for repair when sampling programs of length k. Thus, it converges to effective repair programs with relatively few samples ($k \cdot 10^6$). Like randomization defenses, `DefEnSyn` leverages stochastic noise: it computes a *set* of programs and, at inference, one program from the set is randomly selected. Unlike randomization defenses, this form of stochastic noise has negligible overhead, and it leads to a minor decrease in clean accuracy.

We evaluate `DefEnSyn` on ImageNet classifiers: ConvNeXt [48], DeiT [76], ViT [20], and ResNet-18 [28], consisting of up to 22 million parameters. We consider the more realistic black-box setting, where the attacker has no access to the defended network and can only query it. `DefEnSyn` enhances the robustness of networks against state-of-the-art L_∞ adversarial attacks by +71%, against L_2 attacks by +15%, and against L_0 attacks by +44%. This increase in robust accuracy exceeds the increase of state-of-the-art adversarial training and randomization defenses. `DefEnSyn` also outperforms repair defenses: it increases the robust accuracy of a CIFAR-10 classifier against a backdoor attack [61,46,27] by +27%, compared to +4% obtained by an existing repair [77]. In all experiments, `DefEnSyn` slightly decreases the clean accuracy by about −1%, outperforming existing defenses. It computes repair programs within a few hours, unlike adversarial training, which requires several days. Integrating repair programs into the network poses a negligible overhead during inference: less than $3 \cdot 10^{-5}$ seconds.

In summary, our main contributions are:

- A post-training defense that integrates short repair programs into a network.
- A synthesizer of repair programs that identifies the best neurons for them.
- Extensive evaluation showing that our defense outperforms existing adversarial training, randomization, and repair defenses.

2 Problem Definition

In this section, we define the problem of adversarial defense.

Neural Network Classifiers We focus on classifiers for images. An image is a $d_1 \times d_2$ matrix, consisting of RGB pixels in $[0,1]^3$. A classifier maps an image to a score vector over the classes $[c] = \{1, \ldots, c\}$, i.e., $N : [0,1]^{d_1 \times d_2 \times 3} \to \mathbb{R}^c$. The classification of image x is the class with the highest score, $c' = \text{argmax}(N(x))$. We focus on classifiers implemented by neural networks. Neural networks process data through a series of interconnected layers. Generally, a layer consists of multiple neurons, where a neuron performs some computation (e.g., a weighted sum of its inputs, followed by a non-linear activation function). The exact connections between neurons and the definition of the weights are determined by the network architecture. For example, in a fully connected network [44,75], neurons between two adjacent layers are fully interconnected, and each connection has a unique weight. Other architectures, shown to be highly effective for image processing tasks, are Convolutional Neural Networks (CNNs) [35,65,28,48] and Vision Transformers (ViTs) [20,47,76]. Since the definition of these architectures is not required for understanding our paper, we omit their definition. The effectiveness of a classifier is commonly defined by the *clean accuracy*. This is the percentage of inputs, in a given data set $\mathcal{D}$, that are correctly classified by N:

$$Acc_C(N, \mathcal{D}) = 100 \cdot \frac{\sum_{(x_i,c_i)\in\mathcal{D}} \mathbb{1}\{argmax(N(x_i)) = c_i\}}{|\mathcal{D}|} \tag{1}$$

Attacks An adversarial example attack introduces a small perturbation to a correctly classified input image with the goal of misleading the classifier. Typically, the magnitude of the perturbation is constrained with respect to a chosen norm, such as L_∞, L_2, L_1, or L_0. Formally, given a perturbation bound ϵ and a p-norm, an adversarial attack is a function $\mathcal{A}$ mapping a classifier N, a correctly classified input x and its class c_x to a perturbed image $\mathcal{A}(N, x, c_x) = x' \in [0,1]^{d_1 \times d_2 \times 3}$ such that $\|x - x'\|_p \leq \epsilon$. The attack succeeds if N classifies x' not as c_x (i.e., $\text{argmax}(N(x')) \neq c_x$). This attack is called an *untargeted attack*. Our work can also be extended to targeted attacks, where the goal is that N classifies x' as a target class $c_t \neq c_x$. We note that constraining the attack's perturbation by a single norm is the most widely used attack model; however, there are attack models constraining the perturbation by several norms [32,31,18]. We focus on black-box attacks, where the attacker has no access to the network's internals and can only query the network to obtain the outputs of given inputs (specifically, *score-based attacks*). This setting is more realistic [56,4,40,87], in particular for machine-learning-as-a-service (MLaaS) deployments, where users can only submit queries to the network and observe its outputs (e.g., the scores). Similar to prior work [30,14,40,52], we assume an attacker that adapts its attack by querying the network multiple times. To evaluate the resilience of a classifier against an adversarial attack, it is common to measure the *robust accuracy*

[66,81,1,80,30,15]. The robust accuracy is the percentage of inputs, in a given data set $\mathcal{D}$, that are correctly classified by N when adversarially perturbed:

$$Acc_R(N, \mathcal{D}) = 100 \cdot \frac{\sum_{(x_i,c_i)\in\mathcal{D}} \mathbb{1}\{argmax(N(\mathcal{A}(N, x_i, c_i))) = c_i\}}{|\mathcal{D}|} \qquad (2)$$

Adversarial Defense An adversarial defense is an algorithm whose goal is to increase the robust accuracy of a classifier while maximizing the clean accuracy. Ideally, the goal would be to obtain perfect clean and robust accuracies on any input and for any attack. However, this goal is infeasible for many reasons. For example, some attacks are unknown and some inputs are on the decision boundaries. Instead, the vast majority of adversarial defenses [10,66,39,77,1,15] are given an adversarial attack $\mathcal{A}$ as well as training and test sets $\mathcal{D}_{Tr}, \mathcal{D}_{Ts}$ for evaluating the defended network. Note that $\mathcal{A}$ can be any adversarial attack and can even combine multiple attacks. In this setting, the defense computes a defended network given $2 \cdot |\mathcal{D}_{Tr}|$ requirements (two for each input: classifying the input correctly and classifying its perturbed version by $\mathcal{A}$ correctly) and evaluates the result by measuring the clean and robust accuracies on $\mathcal{D}_{Ts}$. Since these requirements are often conflicting, and since the space of defenses is highly complex, it is common to look for a defense that maximizes the number of satisfied requirements. Formally, an adversarial defense is defined as follows.

Definition 1 (A Defense). *Given a classifier $N : [0, 1]^{d_1 \times d_2 \times 3} \to \mathbb{R}^c$, a data set of image-class pairs $\mathcal{D}_{Tr} \subseteq [0, 1]^{d_1 \times d_2 \times 3} \times [c]$, and an adversarial attack $\mathcal{A}$, a* defense *computes a defended network N' maximizing the robust and clean accuracies $Acc_R(N', \mathcal{D}_{Tr}) + \lambda \cdot Acc_C(N', \mathcal{D}_{Tr})$, where λ is a balancing factor.*

3 Adversarial Defense by Repair Programs

In this section, we describe our defense by a set of *repair programs.*

Our Defense Our defense assumes a network that can be viewed as a composition of an encoder and a classifier, i.e., $N = C \circ E$. Intuitively, E extracts the input's features and passes a feature vector to C to compute the classification. We denote the number of extracted features by m. Formally, $E : \mathbb{R}^{d_1 \times d_2 \times 3} \to \mathbb{R}^m$ and $C : \mathbb{R}^m \to \mathbb{R}^c$. Most popular network architectures can be viewed as a composition of an encoder and a classifier, including fully connected networks, CNNs, and ViTs. Our defense adds a repair program P between the encoder E and classifier C, that is, the defended network is $N' = C \circ P \circ E$. Repair programs are short, and each of their instructions modifies a single entry of the feature vector. Integrating a program into the network is similar to adding a layer to the network, but the program supports more complex computations (e.g., piecewise polynomial functions). Integrating a repairing layer has been proposed [38]; however, only for enforcing linear constraints over the output vector, which cannot express the requirements of clean and robust accuracy. Other defenses that repair a single layer modify an *existing* layer [41,77,25]. While both our approach

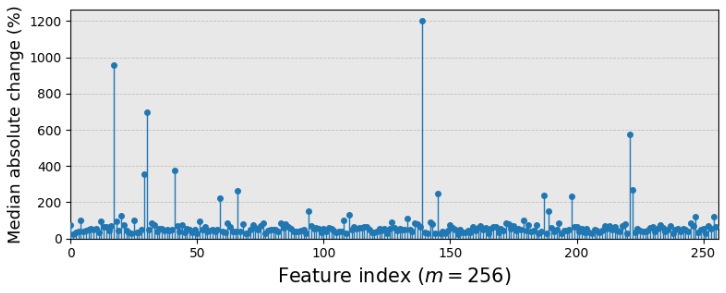


Fig. 2. The median absolute change (%) of every feature of a CIFAR-10 Wide-ResNet classifier following the L_0 `Pixle` attack. Most features are slightly changed.

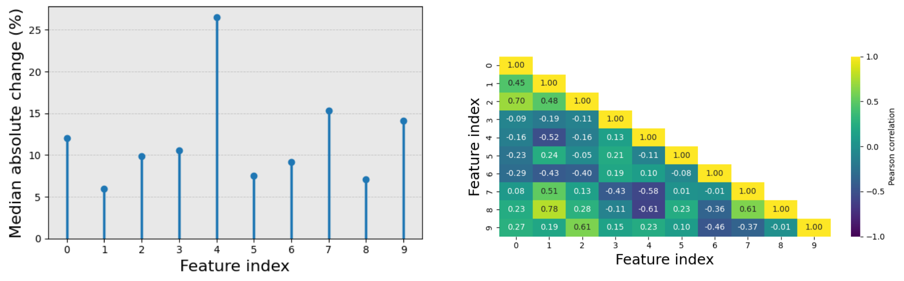


Fig. 3. Left: The median absolute change (%) of the first ten features of a CIFAR-10 Wide-ResNet classifier following the L_∞ `Square` attack. Right: The pairwise Pearson-correlation matrix of these features (computed on clean inputs).

and existing repairs are post-training, our approach is a new form of defense: unlike existing repairs, it does not modify the network parameters or manipulate the network's output. Instead, it *synthesizes a program*, over a (restricted) programming language, which is added *between the network's layers*. This difference is similar to the difference between program repair and program synthesis.

Advantages There are several advantages to repairing the input of the network's classifier part. First, its input dimension tends to be smaller than the input dimensions of previous layers, which reduces the search space of the repair. Second, adversarial attacks tend to change a small subset of these latent features (as demonstrated in Figure 2); thus, identifying and modifying them can mitigate the attack. Third, due to redundancy, features tend to have strong linear correlations with some of the other features [5,6]. This suggests that we can increase the network's robustness without significantly changing its computation by replacing the output of vulnerable features with a linear combination of robust features that have strong correlation with them. To illustrate, we consider a CIFAR-10 [34] classifier and compute the maximal change of its features as a result of the `Square` attack [4]. Figure 3 (left) shows this change for the first ten features (in total there are $m = 256$ features), where feature 4 is one of the top-10 features exhibiting the maximal absolute change. Figure 3 (right) shows the correlation between the first ten features. Features 4 and 8 have a strong negative linear correlation (whereas feature 4 has a lower correlation with the other features). This suggests that replacing the 4^{th} feature (heavily affected by the adversarial attack) with the 8^{th} feature (less affected by the attack) multiplied by a negative constant could mitigate the attack's impact without significantly changing the classifier's computation. Fourth, the synthesis of repair programs does not involve repeatedly running inputs through the full network, but only through the repair program and the classifier part. This reduces the synthesis overhead. Lastly, short repair programs introduce low inference overhead.

Randomization Our defense benefits from randomization without its costs (time overhead and clean accuracy decrease). The motivation for adding randomization is that it challenges attackers in crafting their adversarial examples: despite the black-box access to the network, a fixed defense is known to be more vulnerable, especially if the attacker can pose an unlimited number of queries [82,29]. Thus, our defense generates a *set* of repair programs and at inference, upon every input, randomly selects a program, adds it between the encoder and classifier, and propagates the input through this defended network. This form of stochastic noise enhances the network's robustness with very low overhead, since it only picks one of K programs. Moreover, due to the synthesis, the generated set of repair programs introduces a minor decrease in clean accuracy.

Program Space Our program search space is inspired by fully connected layers. A repair program repairs features (i.e., entries of the feature vector) using arithmetic expressions of (usually other) features. The expressions include standard arithmetic operations (e.g., addition, multiplication), generalizing the weighted sum computed by fully connected layers, and piecewise linear functions (minimum and maximum), generalizing the popular, piecewise linear ReLU activation function [2]. Figure 4 presents the grammar of our repair programs. A repair program P is a sequence of assignment instructions. An assignment A assigns a data element or an operation over two data elements to an entry $\mathbf{v}_i$ of the feature vector ($i \in [m]$). A data element d is a real number r or a feature $\mathbf{v}_i$. An operation Op is an arithmetic operation (addition, subtraction, multiplication, or division) or a comparative function (the minimum or maximum of two data elements). The semantics of repair programs is as expected, except that if a division by zero occurs when running a program, the effect of the respective instruction is removed (i.e., the program runs as if this instruction does not exist).

Example The program: $\boxed{\mathbf{v}_3 = \mathbf{v}_{100} + 3.267;\ \mathbf{v}_{136} = \mathbf{v}_2\ /\ \mathbf{v}_{51};\ \mathbf{v}_{121} = \text{-}3.761}$ has been synthesized for the CIFAR-10 [34] Wide-ResNet-34-10 [85] classifier against the L_2 `Square` attack [4]. It repairs the 3^{rd}, 136^{th}, and 121^{st} entries of the feature vector. The 3^{rd} feature is the sum of the 100^{th} feature and a constant. The 136^{th} feature is the division of the 2^{nd} and 51^{st} features. The 121^{st} feature is set to a constant. All other features are unchanged. Figure 1 shows two more examples.

$$\begin{array}{rrcl}
\textit{(Program)} & P & ::= & A \mid P_1; P_2 \\
\textit{(Assignment)} & A & ::= & \mathbf{v}_i = d \mid \mathbf{v}_i = Op\ (d_1, d_2) \\
\textit{(Operation)} & Op & ::= & add \mid sub \mid mul \mid div \mid max \mid min \\
\textit{(Data Element)} & d & ::= & \mathbf{v}_i \mid r \\
\textit{(Variable)} & \mathbf{v}_i & \in & \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_m\} \\
\textit{(Constant)} & r & \in & \mathbb{R}
\end{array}$$

Fig. 4. The domain-specific language of our repair programs.

4 `DefEnSyn`: A Program Synthesizer of Repair Programs

In this section, we present `DefEnSyn` (**Def**ending by **En**umerative **S**tochastic **Syn**thesis), our synthesizer for computing a set of repair programs as an adversarial defense. We begin by discussing the challenge and our idea. We then explain how `DefEnSyn` computes a set of programs. Lastly, we show its algorithm.

Challenge The goal of `DefEnSyn` is to compute effective repair programs. Given a network $N = C \circ E$ and a training set $\mathcal{D}_{Tr}$, the effectiveness of a program P is the weighted sum of the clean and robust accuracies of the defended network: $score(P) = Acc_R(C \circ P \circ E, \mathcal{D}_{Tr}) + \lambda \cdot Acc_C(C \circ P \circ E, \mathcal{D}_{Tr})$. The higher the score, the more effective the program. Since our program space consists of a small number of operators and since searching for constants in arithmetic expressions over *given* variables is amenable to efficient search procedures (discrete or numerical optimization), we view the main challenge in our program space as selecting the best features to use as variables. The challenge is intensified in our setting for several reasons. First, the features are not standard input variables but rather functions over the network's inputs and some are correlated (Figure 3), which complicates the task of identifying the most suitable features (unlike standard variables which are commonly independent). Further, there is a large set of features (several hundred), which makes this task even more complex. Third, there is no monotonicity between a program's score and the scores of programs that subsume it, which would provide a partial order on the program space and could enable pruning. To the best of our knowledge, identifying an effective small subset of input variables out of a large set of variables, which some are correlated, without a partial order over the programs is a new challenge in program synthesis. To illustrate our problem's complexity, assume a limited program space consisting only of feature assignments $\boxed{\mathbf{v}_{i_1} = \mathbf{v}_{j_1}; \dots; \mathbf{v}_{i_k} = \mathbf{v}_{j_k}}$. Assume that the feature vector dimension is $m = 100$ (in our experiments, $m \in [256, 512]$), and that it takes one millisecond to compute a program's score. It would take about 300 days to enumerate all programs of length three (`DefEnSyn` looks for programs up to length ten). One may consider, as an alternative, stochastic synthesis which learns a distribution over (full) programs to converge to an effective one (e.g., [60,21]). However, even if we check a single program for every possible left-hand side (i.e., without even searching for effective right-hand side expressions), we would need 10^{13} samples for program length ten, which is infeasible.

Key Idea Our key idea is to employ stochastic search to learn the best set of features (as opposed to stochastic synthesis, which learns programs). To cope with the exponential number of possibilities, we learn for each feature *separately* how effective it is as a repaired feature and as a repairing feature. That is, we learn two distributions, each over all m features. We consider the dependencies between the features by (1) defining the effectiveness of a feature to be the average scores of repair programs that include it, and (2) considering programs of increasing lengths, from 1 to k. The ascending program lengths enable us to approximate the effectiveness of a set of features in smaller search spaces. For the sake of explanation, ignore the constants in our language, and assume there are $m = 100$ features and that $k = 10$. In programs of length one, the program space is relatively small: $m \cdot (m + 6 \cdot m^2) = 6{,}010{,}000$ (i.e., the number of possibilities for the left-hand side multiplied by the number of possibilities for the right-hand side, consisting of a single variable or one of our six operators with two operands). Even if we consider constants, 10^6 samples provide good coverage of the programs of length one. Although these programs do not capture dependency between features, we can use the sampled programs' scores to identify effective features for the repair. In programs of length two, the search space is larger: $\approx 6{,}010{,}000^2$ – sampling 10^6 programs covers only $10^{-6}\%$ of the program space. However, if our sampling is biased towards features that are more promising for programs of length one, the search explores programs containing them more frequently. Since there is no monotonicity, some of these promising features may be included in programs with higher scores, while others in programs with lower scores. Accordingly, the learned distributions are updated. Generally, for length i, we sample full programs of length i where the variables are chosen by the learned distributions of the previous program lengths and update the distributions based on these programs' scores. While there is no monotonicity, if a feature is effective for multiple program lengths, it may hint that this feature is effective in longer programs. Thus, when we sample programs of length k, we have a good approximation of the better features for the repair. Consequently, even if we sample a very low number of programs (10^6 is less than $10^{-62}\%$ of the programs of length 10), we can identify effective programs.

Computing a Set of Programs Recall that our defense leverages randomization to enhance robustness and thus `DefEnSyn` generates a set of repair programs. If the synthesized programs are very similar, the randomization would have little effect. Thus, naively returning the K highest-scored programs would be less effective. Instead, `DefEnSyn` forces diversity in the set of programs by requiring that each program has a different set of repaired features. Our diversity definition ignores the right-hand side expressions, including the repairing features, since it may lead to programs with identical repaired features, which will be easier for the attacker to exploit. To compute this set of programs, `DefEnSyn` maintains a dictionary whose keys are sets of repaired features, and the entry for a key is the highest-scored program for this set of repaired features and its score. At the end, `DefEnSyn` returns the K highest-scored repair programs in this dictionary.

Algorithm 1: `DefEnSyn` $(E, C, \mathcal{D}_{Tr}, \mathcal{A})$

```
Input  : Encoder $E$, classifier $C$, training set $\mathcal{D}_{Tr}$, adversarial attack $\mathcal{A}$.
Output: A set of repair programs $\mathcal{P}$.
features = $[E(x_i) \mid (x_i, c_i) \in \mathcal{D}_{Tr}]$
adv_features = $[E(\mathcal{A}(C \circ E, x_i, c_i)) \mid (x_i, c_i) \in \mathcal{D}_{Tr}]$
class = $[c_i \mid (x_i, c_i) \in \mathcal{D}_{Tr}]$
progs = {}                              // The dictionary for the repair programs
$\overline{S}_L = \overline{0}$; $\overline{S}_R = \overline{0}$                 // The average scores of LHS/RHS variables
$S_L = \overline{0}$; $S_R = \overline{0}$                     // The sum of scores of LHS/RHS variables
$Z_L = \overline{0}$; $Z_R = \overline{0}$        // The number of occurrences of LHS/RHS variables
$Dist_{LHS}$ = softmax($\overline{S}_L$); $Dist_{RHS}$ = softmax($\overline{S}_R$)     // The distributions
prev = 0                                          // The sum of the top-$K$ scores
for prog_length = 1 to MAX_NUM_ASSIGNMENTS do
    for $i = 1$ to MAX_ITER_PER_LENGTH do
        $P$ = []                                          // Init a new program
        for length = 1 to prog_length do
            $P$.append(generate_random_assignment($Dist_{LHS}, Dist_{RHS}$))
        score = 0                                    // Init the program's score
        for $j = 1$ to $|\mathcal{D}_{Tr}|$ do
            if argmax($C \circ P$(adv_features[j])) == class[j] then score+ =1

            if argmax($C \circ P$(features[j])) == class[j] then score + = $\lambda$
        lhs_v, rhs_v = extract_vars($P$)
        for $i = 1$ to $m$ do
            if $\mathbf{v}_i \in$ lhs_v then $S_L[i]$ + = score; $Z_L[i]$ + = 1
            if $\mathbf{v}_i \in$ rhs_v then $S_R[i]$ + = score; $Z_R[i]$ + = 1
            $\overline{S}_L[i] = S_L[i]/Z_L[i]$; $\overline{S}_R[i] = S_R[i]/Z_R[i]$
        if lhs_v not in progs.keys() or progs[lhs_v].score < score then
            progs[lhs_v] = ($P$, score)
    $Dist_{LHS}$ = softmax($\overline{S}_L$); $Dist_{RHS}$ = softmax($\overline{S}_R$)
    curr = $\Sigma_{\texttt{score} \in \texttt{topK(progs)}}$score
    if prog_length > 1 and curr/prev < IMPROVE then break
    prev = curr
return top-K programs in progs
```

Synthesis Algorithm 1 shows the algorithm of `DefEnSyn`. Its inputs are a network, given by an encoder E and a classifier C, a training set of images and their classes $\mathcal{D}_{Tr}$, and an adversarial attack $\mathcal{A}$. It returns a set of repair programs $\mathcal{P}$ with different sets of repaired features. `DefEnSyn` begins by computing the feature vectors of the inputs, computing the feature vectors of the adversarial examples, and storing the true classes (Line 1–Line 3). It then initializes the dictionary `progs` mapping a set of repaired features to their best program and score (Line 4). Then, it initializes the vectors $\overline{S}_L$ and $\overline{S}_R$, which store the average scores of each feature on the left-hand and right-hand sides (Line 5). Next, it initializes vectors that enable the computation of these average scores: two vectors for storing the sum of scores S_L and S_R and two vectors for storing the number of feature occurrences Z_L and Z_R (Line 6–Line 7). Afterwards, it initializes the probability distributions to uniform distributions by invoking the softmax operation over the score vectors (Line 8). Next, it initializes the variable `prev` that stores the sum of the scores of the top-K programs, which will be used in an early stopping condition (Line 9). Then, `DefEnSyn` begins synthesizing programs of increasing length from one to `MAX_NUM_ASSIGNMENTS` (Line 10). For every program length, `DefEnSyn` synthesizes `MAX_ITER_PER_LENGTH` programs (Line 11). A program P is synthesized by sampling its instructions using Algorithm 2, given the probability distributions (Line 12–Line 14). Then, it computes the program's score $Acc_R(N', \mathcal{D}_{Tr}) + \lambda \cdot Acc_C(N', \mathcal{D}_{Tr})$, where $N' = C \circ P \circ E$ (Line 15–Line 18). Note that to compute the score of a given input, `DefEnSyn` passes its stored feature vector through the program P and then through the classifier C. That is, every input passes through the encoder E only twice (Line 1–Line 2), rather than for *every* candidate repair program. This significantly reduces the synthesis time, since the encoder consists of most of the layers and thus involves many computations. Then, to update the score vectors and `progs`, `DefEnSyn` extracts from P the features on the left-hand side and on the right-hand side (Line 19). Accordingly, it updates the average vectors (Line 20–Line 23) and updates `progs` if P's set of repaired features is new or if its score is better (Line 24–Line 25). After synthesizing all programs for a given length, `DefEnSyn` updates the probability distributions using the softmax operation (Line 26). Then, it checks whether it can stop early by checking the improvement of this iteration (Line 27–Line 29). This is checked by computing the ratio of the sum of the scores of the current top-K programs and this sum in the previous iteration. If the ratio is below a threshold `IMPROVE`, `DefEnSyn` terminates. Otherwise, it updates `prev` and continues to another iteration. At the end, `DefEnSyn` returns the top-K programs in `progs`, to obtain a diverse set of repair programs. We note that, in practice, synthesizing effective repair programs can be achieved with a small training set $\mathcal{D}_{Tr}$ (several hundred suffice). In case the network's original training set is unavailable, it is possible to synthesize a training set, as suggested by [56,42,86].

Sampling Instructions Algorithm 2 generates a random instruction. Its inputs are the probability distributions for selecting the features for the left-hand side and the right-hand side. It first samples the candidates for operands: the feature to repair, the two repairing features, and a real-valued constant (within a predetermined interval $[l, u]$). It then determines the first and second operands (`d1` and `d2`) by uniformly sampling from the two repairing features and the constant. Next, it uniformly samples an operator. Lastly, it uniformly samples the right-hand side over the constant, the first repairing feature, and the arithmetic computation. This form of sampling expresses our preference for simple assignments over arithmetic operations, since they provide simpler repairs.

Algorithm 2: generate_random_assignment($Dist_{LHS}$, $Dist_{RHS}$)

```
input  : $Dist_{LHS}, Dist_{RHS} \in [0,1]^m$, probability distributions over the
         features for the left-hand side and the right-hand side.
output: An assignment instruction.
1 lhs_v $\sim Dist_{LHS}(\{\mathbf{v}_1, \dots, \mathbf{v}_m\})$
2 rhs_v1 $\sim Dist_{RHS}(\{\mathbf{v}_1, \dots, \mathbf{v}_m\})$
3 rhs_v2 $\sim Dist_{RHS}(\{\mathbf{v}_1, \dots, \mathbf{v}_m\})$
4 const $\sim$ Uniform($[l, u]$)
5 d1 $\sim$ Uniform({const, rhs_v1, rhs_v2})
6 d2 $\sim$ Uniform({const, rhs_v1, rhs_v2})
7 op $\sim$ Uniform({+, -, /, •, max, min})
8 rhs $\sim$ Uniform({const, rhs_v1, op(d1,d2)})
9 return $\boxed{lhs\_v = rhs}$
```

A Running Example Figure 5 shows a running example of `DefEnSyn`. Its inputs are a network split into an encoder E and a classifier C, a training set of three images, and an attack $\mathcal{A}$. `DefEnSyn` begins by iterating over the training set and computing for each image its perturbed image using $\mathcal{A}$. It then passes the image and the perturbed image through E and stores the feature vectors in `features` and `adv_features` (Figure 5(a)). In this example, there are three features ($\mathbf{v} \in \mathbb{R}^3$). Then, `DefEnSyn` enumerates programs by incrementally increasing the programs' lengths, from one to `MAX_NUM_ASSIGNMENTS` $= 10$. For each length, it generates `MAX_ITER_PER_LENGTH` $= 10^6$ programs. For program length 1, the left-hand side $Dist_{LHS}$ and the right-hand side $Dist_{RHS}$ distributions are uniform: $(1/3, 1/3, 1/3)$. After generating 10^6 programs of length 1, $Dist_{LHS}$ and $Dist_{RHS}$ are updated based on their scores (Figure 5(b), top left). It then continues to program length 2 and generates 10^6 programs. Figure 5(b) shows the last iteration of `prog_length` $= 2$ (i.e., $i = 10^6$). In this iteration, the sampled program is $\boxed{\text{P: } \mathbf{v}_1 = \mathbf{v}_3 + 3.5;\ \mathbf{v}_2 = -1.7}$. It is constructed by calling Algorithm 2 twice with $Dist_{LHS}$ and $Dist_{RHS}$. `DefEnSyn` computes P's score by passing each feature vector and each adversarial feature vector through P and then C. For each, it checks whether the predicted class is correct. If so, the score increases by $\lambda = 1$ for a feature vector and by 1 for an adversarial feature vector. In this example, the score is 4. Then, for each LHS feature, $\mathbf{v}_1$ and $\mathbf{v}_2$, it adds 1 to its entry in Z_L and adds the score to its entry in S_L. Similarly, it updates the entry of the RHS feature $\mathbf{v}_3$ in Z_R and S_R. Then, `DefEnSyn` checks the dictionary `progs` at the entry of the repaired feature set $\{\mathbf{v}_1, \mathbf{v}_2\}$. In this example, 4 is a better score and thus `progs` is updated (Figure 5(b), right). After this iteration, which completes the sampling for length 2, it updates $Dist_{LHS}$ and $Dist_{RHS}$ given S_L, S_R, Z_R, and Z_L. For instance, the probability of selecting $\mathbf{v}_1$ for the left-hand side rises from 0.1 to 0.2, whereas that of $\mathbf{v}_2$ drops from 0.7 to 0.6 (Figure 5(c), left). `DefEnSyn` next checks whether the program length 2 yields a sufficient improvement in the cumulative score of the top-K programs. In this example, it does, and so it proceeds to length 3. Figure 5(c) shows its first itera-

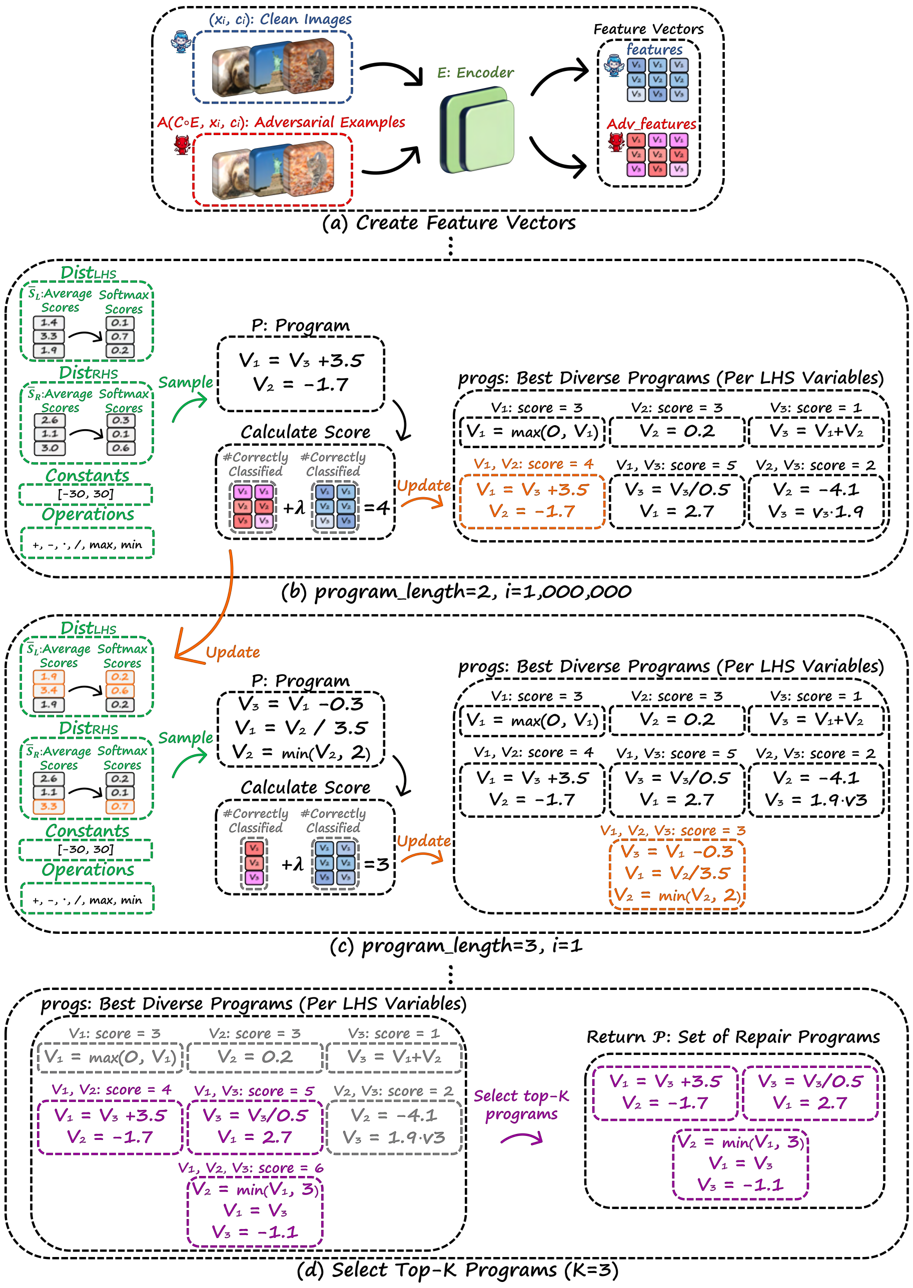


Fig. 5. Illustration of `DefEnSyn`.

tion, where $\boxed{\text{P: } \mathbf{v}_3 = \mathbf{v}_1 - 0.3;\ \mathbf{v}_1 = \mathbf{v}_2/3.5;\ \mathbf{v}_2 = min(\mathbf{v}_2, 2)}$ and its score is 3. It updates S_L, S_R, Z_R, Z_L and then updates `progs`, since P is the first program with left-hand side $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$ (Figure 5(c), right). When `DefEnSyn` completes, it returns the top-K (in the example, $K = 3$) programs with the highest scores in `progs`, each repairing a different set of features (Figure 5(d)).

Limitations `DefEnSyn` has two main limitations. First, it is an empirical defense and does not guarantee soundness or completeness, i.e., the synthesized programs may not always ensure robustness, and `DefEnSyn` may miss programs that could ensure robustness. Second, to keep the synthesis overhead tractable, it does not analyze dependencies among features or relations between the repair programs.

5 Evaluation

In this section, we evaluate `DefEnSyn` and show: (1) it improves the robustness of networks against state-of-the-art black-box L_∞, L_2, and L_0 adversarial attacks, outperforming state-of-the-art adversarial training and randomization defenses, (2) it only slightly reduces the clean accuracy of the defended networks, with a maximum decrease of -2%, (3) it can synthesize effective repair programs within a few hours, unlike existing adversarial training posing a training overhead of 1-2 days, (4) the inference overhead stemming from incorporating its repair programs into a network is negligible, (5) it effectively counters backdoor attacks, outperforming an existing repair [77], and (6) its synthesis components are important for achieving both high clean accuracy and high robust accuracy.

Implementation and Setup We implemented `DefEnSyn`[1] in Python using PyTorch. Our implementation supports GPU parallelization. Experiments ran on an Ubuntu 20.04.2 OS on a dual AMD EPYC 7742 server with 1TB RAM and eight NVIDIA A100 GPUs. Unless otherwise stated, the hyper-parameters are $|\mathcal{D}_{Tr}| = 750$, `MAX_NUM_ASSIGNMENTS` = 10, `MAX_ITER_PER_LENGTH` = 10^6, `IMPROVE` = 1.01, $\lambda = 1$, $K = 30$, and $[l, u] = [-30, 30]$ (Algorithm 2). We evaluate `DefEnSyn` on two image data sets, CIFAR-10 [34] and ImageNet [19], consisting of $d \times d \times 3$ colored images, where $d = 32$ for CIFAR-10 and $d = 224$ for ImageNet. An image is classified as one of 10 classes for CIFAR-10 and as one of 1,000 classes for ImageNet. We consider different networks, where for each, we consider as the encoder all layers but the last one and as the classifier the last layer. The feature vector dimension m is between 256 and 512. We measure the clean and robust accuracies over inputs in the test sets of CIFAR-10 or ImageNet, which are disjoint from the training sets `DefEnSyn` uses.

5.1 Defense against L_∞, L_2, and L_0 Adversarial Example Attacks

We next evaluate `DefEnSyn`'s effectiveness for black-box adversarial example attacks. We compare `DefEnSyn` with several state-of-the-art defenses to assess its

[1] https://github.com/TomYuviler/DefEnSyn

effectiveness. The selected baselines represent diverse approaches in adversarial defense. The first baseline is the adversarial training proposed by [66], denoted as `AdvTrain`, which is the current state-of-the-art defense against L_∞ adversarial attacks. The second baseline is the diffusion-based randomization defense introduced by [10], denoted as `DiffRandom`. This approach utilizes a pre-trained denoising diffusion model to modify the inference phase without additional training or synthesis and stands as the state-of-the-art randomization-based defense against L_2 adversarial attacks. The third baseline is the randomized smoothing technique developed by [15], which incorporates Gaussian noise during both training and inference phases to defend against L_2 adversarial attacks, denoted as `GausRandom`. The fourth baseline is the randomization-based defense customized for L_0 attacks, proposed by [39], which performs a randomized ablation of image pixels. This defense is the state-of-the-art defense against L_0 attacks and is denoted as `AblaRandom`. For all baselines, we use the authors' code with their default hyper-parameters. In this experiment, we focus on large networks for which no scalable repair defense currently exists against adversarial attacks.

Evaluated Networks We consider four network classifiers for ImageNet. The first network, denoted as `ConvNeXt`, is Isotropic ConvNeXt-S [48], one of the state-of-the-art models for ImageNet classification using convolutional neural networks, with 22 million parameters. The second network, denoted as `DeiT`, is Data-Efficient Image Transformer [76], which is a Vision Transformer model [20] with an efficient training process and 22 million parameters. The third network, denoted as `ViT`, is a Vision Transformer model [20], with 6 million parameters. The fourth network, denoted as `ResNet18`, is ResNet-18 [28], which is a convolutional neural network with 12 million parameters. The weights for `ConvNeXt` were obtained from the original ConvNeXt repository[2]. The weights of the other networks were obtained from `timm` [78], an open-source collection of classifiers.

Adversarial Example Attacks We evaluate `DefEnSyn` against the `Square` attack [4], which is the state-of-the-art black-box adversarial example attack (based on a recent benchmark [87]), for the L_∞ and L_2 norms. The perturbation limits are $\epsilon = 4/255$ for the L_∞ norm and $\epsilon = 0.5$ for the L_2 norm. For the L_0 norm, we evaluate `DefEnSyn` against two state-of-the-art black-box attacks: the `Pixle` attack [57] and the `Sparse-RS` attack [16]. For the `Square` and the `Pixle` attacks, we use the implementation in Torchattacks [33], and for the `Sparse-RS` attack, we use the authors' code. For all attacks, we use the default hyper-parameters.

Results We run the attacks on the defended networks over 10,000 randomly selected images from ImageNet's test set. Table 2 reports the clean accuracy, robust accuracy, training/synthesis overhead, and inference time. The results show that `DefEnSyn`'s repair programs significantly increase the models' robust accuracy against the L_∞ `Square` attack by $+70.6\%$, with a minor decrease in clean accuracy of less than -1%. In contrast, `AdvTrain`, the state-of-the-art

[2] `https://github.com/facebookresearch/ConvNeXt`

Table 2. `DefEnSyn` vs. state-of-the-art adversarial training and randomization defenses.

Classifier (# Params.)	Attack	Defense	Clean acc. (%)	Robust acc. (%)	Training/synthesis overhead (H)	Inference time (S)
`ConvNeXt` (22M)	`Square` (L_∞)	`No defense`	79.72	7.69	-	2.9×10^{-5}
		`AdvTrain`	66.11	51.97	49.4	2.9×10^{-5}
		`DefEnSyn`	79.52	77.38	1.0	2.9×10^{-5}
`DeiT` (22M)	`Square` (L_∞)	`No defense`	79.21	4.99	-	3.1×10^{-5}
		`AdvTrain`	67.23	54.36	45.2	3.1×10^{-5}
		`DefEnSyn`	78.77	76.44	0.9	3.2×10^{-5}
`ResNet18` (12M)	`Square` (L_2)	`No defense`	71.36	55.97	-	2.9×10^{-5}
		`GausRandom`	50.08	49.61	31.8	0.17
		`DefEnSyn`	71.15	70.96	2.2	2.9×10^{-5}
`ViT` (6M)	`Square` (L_2)	`No defense`	69.33	54.21	-	2.9×10^{-5}
		`DiffRandom`	62.86	62.86	0	59.75
		`DefEnSyn`	69.28	69.21	1.5	3.1×10^{-5}
`ResNet18` (12M)	`Pixle` (L_0)	`No defense`	71.04	10.71	-	2.9×10^{-5}
		`AblaRandom`	35.13	32.53	26.3	6.89
		`DefEnSyn`	69.98	40.75	1.5	3.2×10^{-5}
`ResNet18` (12M)	`Sparse-RS` (L_0)	`No defense`	71.79	6.56	-	2.9×10^{-5}
		`AblaRandom`	35.73	34.47	26.3	6.89
		`DefEnSyn`	70.46	65.0	2.1	3.1×10^{-5}

adversarial training, increases the models' robust accuracy by +46.9% against the L_∞ `Square` attack, with a decrease in clean accuracy of −12.4%. For the L_2 `Square` attack, `DefEnSyn` increases the robust accuracy by +15.0%, with a minor decrease in clean accuracy of −1%. In contrast, the state-of-the-art randomization-based defenses, `GausRandom` and `DiffRandom`, on average increase the robust accuracy by +1.2%, with a significant decrease in clean accuracy of −14%. For the L_0 attacks, `DefEnSyn` increases the robust accuracy by +44.2%, with a minor decrease in the clean accuracy of −1%. In contrast, `AblaRandom`, the state-of-the-art defense for L_0 attacks, enhances the robust accuracy by +24.9% and significantly decreases the clean accuracy by −36%. Additionally, on average, `DefEnSyn`'s synthesis overhead is 1.5 hours, while the average training overhead of `AdvTrain` is 47.3 hours, of `GausRandom` is 31.8 hours, and of `AblaRandom` is 26.3 hours. The `DiffRandom` defense does not require any training, but it assumes access to a pre-trained denoiser, unlike `DefEnSyn` and the other baselines. The average inference time per image of `DefEnSyn` and `AdvTrain` is similar to that of the original (undefended) network: approximately 3×10^{-5} seconds. In contrast, the inference time of the randomization-based techniques is significantly higher and can reach up to one minute in the worst case.

5.2 Defense against Backdoor Attacks

Next, we compare `DefEnSyn` to a repair defense against a backdoor attack [61,46,27]. Backdoor attacks embed malicious behaviors into a classifier

Table 3. `DefEnSyn` vs. NNRepair on a CIFAR-10 classifier against a backdoor attack.

Classifier (# Params.)	Attack	Defense	Clean acc. (%)	Robust acc. (%)
`ConvCIFAR` (0.9M)	Backdoor	`No defense`	72.26	15.89
		`NNRepair-I`	72.28	16.70
		`NNRepair-L`	71.65	19.66
		`DefEnSyn-30`	70.29	34.18
		`DefEnSyn-1`	70.69	43.06

during its training phase by introducing a small proportion of poisoned data into the training set, each containing a specific pattern ("trigger"), such as a small white square. The model, once trained with this poisoned data set, behaves normally on standard inputs but produces a specific incorrect output when the trigger is present in the input. We note that, unlike the majority of adversarial example attacks, the perturbation is constant and is not influenced by the original input image.

Setting We compare to NNRepair [77], which utilizes a constraint solver to make minor adjustments to the weights of specific network layers to mitigate the backdoor attack's effects. We consider two variants of NNRepair: `NNRepair-I`, which modifies an intermediate layer, and `NNRepair-L`, which modifies the last layer. We evaluate `DefEnSyn` and NNRepair on the convolutional classifier for CIFAR-10, `ConvCIFAR`, evaluated by NNRepair[3]. This network has 890,000 parameters and achieves a test accuracy of 72.26% on clean inputs. However, its accuracy drops significantly to 15.89% for inputs with the trigger of a 3×3 white square positioned at the bottom-right corner of the image. The defense's objective is to enhance the classification accuracy for inputs with the trigger without decreasing the clean accuracy. Since the attack is independent of the input image, introducing randomness into the network as a defense mechanism is unnecessary. Thus, in this experiment, we consider a variant of `DefEnSyn` that defends using the repair program with the highest score seen during the synthesis process, instead of defending using a set $\mathcal{P}$ of K=`30` repair programs. We denote `DefEnSyn`'s defense by `DefEnSyn-30` and its variant by `DefEnSyn-1`.

Results We run a backdoor attack on all 10,000 images from CIFAR-10's test set. The attack adds the trigger: a 3×3 white square placed at the bottom-right corner. Table 3 shows the results. While both approaches reduce the network's clean accuracy by at most -2%, `DefEnSyn` significantly enhances the robust accuracy: by $+18.29\%$ when K=`30` and by $+27.17\%$ when K=`1`. In contrast, NNRepair enhances the robust accuracy of the network by at most $+3.77\%$.

[3] `https://github.com/nnrepair`

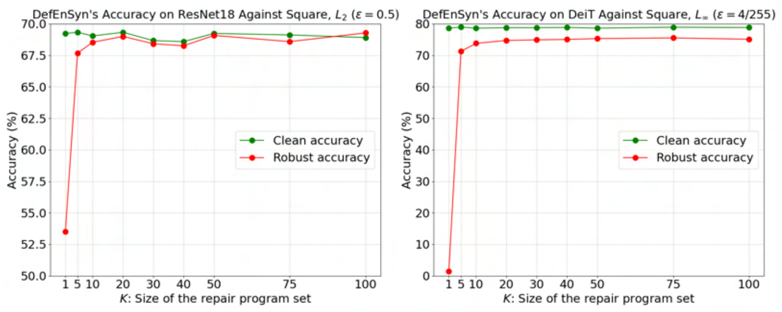


Fig. 6. The impact of different values of K (top-K) on the robust and clean accuracies.

5.3 Ablation Study

Lastly, we provide an ablation study showing the importance of: (1) defending by a *set* of repair programs, (2) synthesizing programs and sampling variables from learned distributions (showing that merely hiding the chosen repair program from the attacker is not enough), and (3) allowing multiple instructions in the repair programs (justifying our large program space). In addition, we analyze the impact that key hyper-parameters have on the performance of `DefEnSyn`.

Set of Programs We evaluate the importance of defending using a set of repair programs by running `DefEnSyn` with different values of K (i.e., the size of the repair program set $\mathcal{P}$). We focus on two ImageNet classifiers, `ResNet18` and `DeiT`, and the `Square` attack, which runs over 10,000 random images from ImageNet's test set. We defend the first model against the L_2 `Square` attack and the second model against the L_∞ `Square` attack. Figure 6 shows the results. It shows that with $K = 1$ (a single repair program), the robust accuracy does not increase. This is expected since the attacker can query the network multiple times (in a black-box fashion) to adapt its attack to the defense. As K increases, the robust accuracy increases and it stabilizes around $K \geq 20$. Results also show that the value of K has a low impact on the clean accuracy.

Synthesis and Learned Distributions Next, we compare `DefEnSyn` to two variants: `Random` and `Uniform-DefEnSyn`. `Random` generates a random program upon each input during inference (i.e., it randomly selects a program length (1–10) and generates instructions using Algorithm 2 with uniform probability distributions). `Uniform-DefEnSyn` runs the synthesis (Algorithm 1) but does not update the variable distributions, i.e., $Dist_{LHS}$ and $Dist_{RHS}$ remain uniform distributions. We compare the three algorithms on `ResNet18` for the L_∞ `Square` attack, with a perturbation limit of $\epsilon = 4/255$, and 10,000 random images from ImageNet's test set. Table 4 presents the results. The results show that `Random` improves the robust accuracy by +21%, but decreases the clean accuracy significantly by −44.8%. `Uniform-DefEnSyn` increases the robust accuracy by +54.6%, but decreases the clean accuracy by −2%. `DefEnSyn` increases the robust accu-

Table 4. Effectiveness of `DefEnSyn`'s synthesis and learned distributions.

Defense	Clean accuracy (%)	Robust accuracy (%)
`No defense`	71.88	5.21
`Random`	27.09	26.21
`Uniform-DefEnSyn`	69.90	59.79
`DefEnSyn`	70.83	65.00

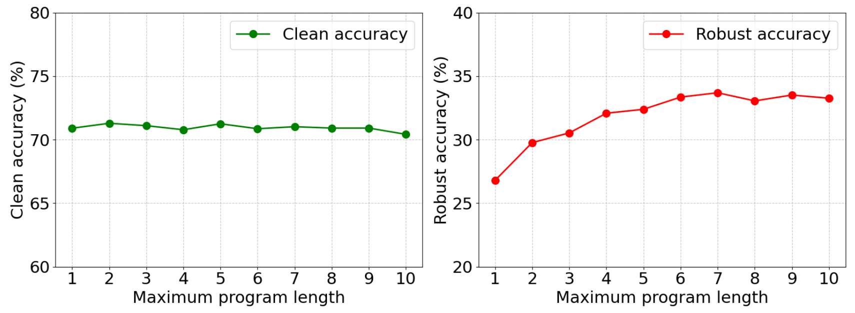


Fig. 7. Clean accuracy (left) and robust accuracy (right) for different program lengths.

racy by +59.8% and decreases the clean accuracy by only −1%. This shows the importance of our synthesis and learned distributions.

Multiple Instructions Next, we show the importance of `DefEnSyn`'s ability to synthesize repair programs with multiple instructions. We focus on the CIFAR-10 classifier and the backdoor attack (Section 5.2), evaluated over all 10,000 images from CIFAR-10's test set. We run `DefEnSyn` and report the clean and robust accuracies of the top-K (for $K = 30$) repair programs after each iteration of `prog_length`. Figure 7 shows the results. The results indicate that the synthesis of multiple instructions allows `DefEnSyn` to increase the robust accuracy.

Training Set Size We next evaluate the impact of the size of the training set $\mathcal{D}_{Tr}$ by running `DefEnSyn` with training sets of different sizes. We focus on the widely known Wide-ResNet-28-4 CIFAR-10 classifier [85], with 6 million parameters, and the L_∞ `Square` attack, with a perturbation limit of $\epsilon = 8/255$. We evaluate `DefEnSyn`'s defense over all 10,000 images from the CIFAR-10 test set. Figure 8 shows the clean and robust accuracies as a function of the training set size. It shows that even a very small training set of 10 images enhances the robust accuracy by almost +20%, with a decrease of −3% in the clean accuracy. With 50 images, the robust accuracy increases by an additional +2%, and the decrease in clean accuracy drops to −1%. The robust and clean accuracies slightly improve with larger training sets and stabilize when the training set size is 250.

Constant Interval We next analyze the impact of the size of the interval $[l, u]$, which defines the range of the constants in the synthesized repair pro-

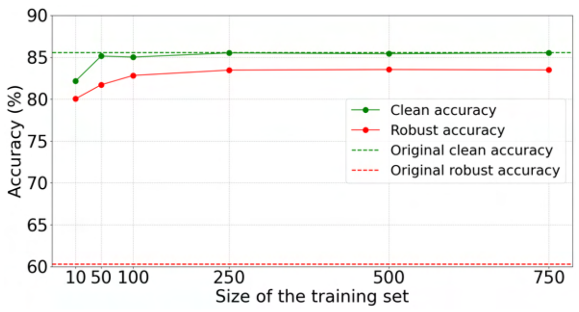


Fig. 8. The impact of the training set size $|\mathcal{D}_{Tr}|$ on the robust and clean accuracies.

Table 5. The impact of the constant interval $[l, u]$ on the robust and clean accuracies.

$[l, u]$	Clean accuracy (%)	Robust accuracy (%)
$[-5, 5]$	85.22	83.55
$[-15, 15]$	85.39	83.57
$[-30, 30]$	85.53	83.55
`No defense`	85.59	60.30

grams (Algorithm 2). We run `DefEnSyn` with different intervals $[l, u]$ where $[l, u] \in \{[-5, 5], [-15, 15], [-30, 30]\}$. We focus on the Wide-ResNet-28-4 CIFAR-10 classifier and the L_∞ `Square` attack with a perturbation limit of $\epsilon = 8/255$. We evaluate `DefEnSyn`'s defense over all 10,000 images from the CIFAR-10 test set. Table 5 shows the clean and robust accuracies. The results show that the clean and robust accuracies are very similar, suggesting that further increasing the interval is unnecessary.

Balancing Factor Lastly, we analyze the impact of the balancing factor λ, which weights the importance of the clean accuracy relative to the robust accuracy in the program's score function. We run `DefEnSyn` with different values of λ on the Wide-ResNet-28-4 CIFAR-10 classifier and the L_∞ `Square` attack with a perturbation limit of $\epsilon = 8/255$. We evaluate `DefEnSyn`'s defense over all 10,000 images from the CIFAR-10 test set. Table 6 shows the clean and robust accuracies. The results show the expected trade-off: higher values of λ lead to better clean accuracy at the expense of robust accuracy, while lower values improve robustness but cause a decrease in clean accuracy.

6 Related Work

In this section, we discuss the closest related work.

Neural Network Repair Several repair techniques have been proposed for neural networks. NNRepair enhances robustness by modifying the weights in a single

Table 6. The impact of the balancing factor λ on the robust and clean accuracies.

λ	Clean accuracy (%)	Robust accuracy (%)
0	84.09	83.59
0.5	85.39	83.57
1	85.53	83.55
5	85.58	81.99
100	85.59	77.19
`No defense`	85.59	60.30

layer using constraint solvers [77]. MODE performs model state differential analysis to pinpoint network parameters responsible for undesired behaviors, followed by network retraining with selected inputs [49]. REGLO applies a verification-guided algorithm to detect and rectify violations of the global robustness property by adjusting the weights in the network's last layer [24]. CARE repairs multiple layers via causality-based fault localization, targeting weight adjustments in neurons linked to undesired behavior [72]. Several works identify the minimal weight modifications necessary to modify the network's behavior [59,25,69]. Arachne repairs weights via Differential Evolution [67]. APRNN provides a provable repair [74], looking for a repair that is correct for all inputs within a convex bounded polytope. Several works repair via architectural modifications, such as integrating a compact patch network [23], adding a self-correcting layer [38], and decoupling the network into separate activation and value networks [70].

Randomization Defenses Like `DefEnSyn`, several defenses incorporate randomness into the classification process. One defense modifies the classification by randomly selecting multiple samples and then predicting the class based on a majority vote among these samples [9]. Others rely on randomized smoothing, wherein Gaussian noise layers are added to the model, which is then trained with these layers [37,15,45]. During inference, multiple copies of the original input are generated, and the prediction relies on the average of the predictions. Another approach modifies only the inference procedure [62,10]. Given a (pre-trained) classifier, they create Gaussian noise-corrupted copies of the input image, denoise them, and then base the classification on the majority vote across all denoised images. Others use a diffusion model to introduce random noise into an adversarial example and then employ a reverse denoising process to purify the image before classification [53]. A different approach defends against L_0 adversarial example attacks using randomized ablation of pixels [39].

Program Synthesis Like `DefEnSyn`, several works guide a synthesizer by learning. Probe introduces just-in-time learning, which guides the search by leveraging partial solutions (programs) and probabilistic models [7]. Counterexample Guided Inductive Synthesis (CEGIS) prunes the search space using counterexamples [68]. Divide-and-conquer independently enumerates smaller expressions that are suitable for specific subsets of inputs [3]. Large language models (LLMs) have been shown to be effective in directing the synthesizer through an iterative feedback loop between the LLM and the synthesizer [43]. Some works employ Bayesian methods to adjust the sampling distributions for programs [60,21], while others rely on Markov Chain Monte Carlo to synthesize programs [63].

7 Conclusion

We present `DefEnSyn`, a synthesizer of repair programs for enhancing the robustness of neural network classifiers. A repair program repairs a few neurons using other neurons. `DefEnSyn` performs a stochastic search to identify effective neurons for the repair. To scale, it samples programs of increasing lengths. `DefEnSyn` synthesizes a set of diverse repair programs so that, at inference, a program can be randomly selected for the repair. We evaluate `DefEnSyn` on large networks for ImageNet and CIFAR-10 against black-box adversarial example attacks. The evaluation shows that `DefEnSyn`'s repair programs enhance the networks' robust accuracy on average by +71% against L_∞ attacks, by +15% against L_2 attacks, by +44% against L_0 attacks, and by +27% against backdoor attacks. `DefEnSyn` decreases the clean accuracy by approximately −1%. These accuracies exceed those of state-of-the-art customized defenses relying on adversarial training, randomization, or repair. `DefEnSyn`'s synthesis overhead is a few hours, and its inference overhead is negligible.

References

1. Addepalli, S., Jain, S., Sriramanan, G., Babu, R.V.: Scaling adversarial training to large perturbation bounds. In ECCV (2022), `https://doi.org/10.1007/978-3-031-20065-6_18`
2. Agarap, A.F.: Deep learning using rectified linear units (relu). CoRR **abs/1803.08375** (2018), `http://arxiv.org/abs/1803.08375`
3. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In TACAS (2017), `https://doi.org/10.1007/978-3-662-54577-5_18`
4. Andriushchenko, M., Croce, F., Flammarion, N., Hein, M.: Square attack: A query-efficient black-box adversarial attack via random search. In ECCV (2020), `https://doi.org/10.1007/978-3-030-58592-1_29`
5. Ayinde, B.O., Inanc, T., Zurada, J.M.: On correlation of features extracted by deep neural networks. In IJCNN (2019), `https://doi.org/10.1109/IJCNN.2019.8852296`
6. Ayinde, B.O., Inanc, T., Zurada, J.M.: Redundant feature pruning for accelerated inference in deep neural networks. In Neural Networks **118** (2019), `https://doi.org/10.1016/j.neunet.2019.04.021`
7. Barke, S., Peleg, H., Polikarpova, N.: Just-in-time learning for bottom-up enumerative synthesis. In OOPSLA (2020), `https://doi.org/10.1145/3428295`
8. Bornholt, J., Torlak, E., Grossman, D., Ceze, L.: Optimizing synthesis with metasketches. In POPL (2016), `https://doi.org/10.1145/2837614.2837666`

9. Cao, X., Gong, N.Z.: Mitigating evasion attacks to deep neural networks via region-based classification. In ACSAC (2017), `https://doi.org/10.1145/3134600.3134606`
10. Carlini, N., Tramèr, F., Dvijotham, K.D., Rice, L., Sun, M., Kolter, J.Z.: (certified!!) adversarial robustness for free! In ICLR (2023), `https://openreview.net/pdf?id=JLg5aHHv7j`
11. Carlini, N., Wagner, D.A.: Towards evaluating the robustness of neural networks. In IEEE Symposium on Security and Privacy, SP (2017), `https://doi.org/10.1109/SP.2017.49`
12. Chen, P., Sharma, Y., Zhang, H., Yi, J., Hsieh, C.: EAD: elastic-net attacks to deep neural networks via adversarial examples. In AAAI (2018), `https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16893`
13. Chen, Q., Lamoreaux, A., Wang, X., Durrett, G., Bastani, O., Dillig, I.: Web question answering with neurosymbolic program synthesis. In PLDI (2021), `https://doi.org/10.1145/3453483.3454047`
14. Chen, S., Carlini, N., Wagner, D.A.: Stateful detection of black-box adversarial attacks. CoRR **abs/1907.05587** (2019), `http://arxiv.org/abs/1907.05587`
15. Cohen, J., Rosenfeld, E., Kolter, J.Z.: Certified adversarial robustness via randomized smoothing. In ICML (2019), `http://proceedings.mlr.press/v97/cohen19c.html`
16. Croce, F., Andriushchenko, M., Singh, N.D., Flammarion, N., Hein, M.: Sparse-rs: A versatile framework for query-efficient sparse black-box adversarial attacks. In AAAI (2022), `https://doi.org/10.1609/aaai.v36i6.20595`
17. Croce, F., Hein, M.: Mind the box: l_1-apgd for sparse adversarial attacks on image classifiers. In ICML (2021), `http://proceedings.mlr.press/v139/croce21a.html`
18. Croce, F., Hein, M.: Adversarial robustness against multiple and single l_p-threat models via quick fine-tuning of robust classifiers. In ICML (2022), `https://proceedings.mlr.press/v162/croce22b.html`
19. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In CVPR (2009), `https://doi.org/10.1109/CVPR.2009.5206848`
20. Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., Houlsby, N.: An image is worth 16x16 words: Transformers for image recognition at scale. In ICLR (2021), `https://openreview.net/forum?id=YicbFdNTTy`
21. Ellis, K., Solar-Lezama, A., Tenenbaum, J.: Sampling for bayesian program learning. In NIPS (2016), `https://proceedings.neurips.cc/paper/2016/hash/afd4836712c5e77550897e25711e1d96-Abstract.html`
22. Fiscutean, A.: How the eu ai act regulates artificial intelligence: What it means for cybersecurity (2023), `https://www.csoonline.com/article/1258597`
23. Fu, F., Li, W.: Sound and complete neural network repair with minimality and locality guarantees. In ICLR (2022), `https://openreview.net/forum?id=xS8AMYiEav3`
24. Fu, F., Wang, Z., Fan, J., Wang, Y., Huang, C., Chen, X., Zhu, Q., Li, W.: REGLO: Provable neural network repair for global robustness properties. In NeurIPS Workshop (2022), `https://openreview.net/forum?id=FRTXdodwsoA`
25. Goldberger, B., Katz, G., Adi, Y., Keshet, J.: Minimal modifications of deep neural networks using verification. In LPAR (2020), `https://doi.org/10.29007/699q`
26. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In ICLR (2015), `http://arxiv.org/abs/1412.6572`

27. Gu, T., Dolan-Gavitt, B., Garg, S.: Badnets: Identifying vulnerabilities in the machine learning model supply chain. CoRR **abs/1708.06733** (2017), http://arxiv.org/abs/1708.06733
28. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In CVPR (2016), https://doi.org/10.1109/CVPR.2016.90
29. He, Z., Rakin, A.S., Fan, D.: Parametric noise injection: Trainable randomness to improve deep neural network robustness against adversarial attack. In CVPR (2019), http://openaccess.thecvf.com/content_CVPR_2019/html/He_Parametric_Noise_Injection_Trainable_Randomness_to_Improve_Deep_Neural_Network_CVPR_2019_paper.html
30. Hung-Quang, N., Lao, Y., Pham, T., Wong, K., Doan, K.D.: Understanding the robustness of randomized feature defense against query-based adversarial attacks. In ICLR (2024), https://openreview.net/forum?id=vZ6r9GMT1n
31. Jiang, E., Singh, G.: RAMP: boosting adversarial robustness against multiple l_p perturbations for universal robustness. In NeurIPS (2024), http://papers.nips.cc/paper_files/paper/2024/hash/4d5ce4a7ebf588834db127965cdb5ccb-Abstract-Conference.html
32. Jiang, E., Singh, G.: Towards universal certified robustness with multi-norm training. CoRR **abs/2410.03000** (2024), https://doi.org/10.48550/arXiv.2410.03000
33. Kim, H.: Torchattacks: A pytorch repository for adversarial attacks. arXiv preprint arXiv:2010.01950 (2020), https://github.com/Harry24k/adversarial-attacks-pytorch
34. Krizhevsky, A.: Learning multiple layers of features from tiny images (2009), https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf
35. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In NeurIPS (2012), https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html
36. Kurakin, A., Goodfellow, I.J., Bengio, S.: Adversarial examples in the physical world. In ICLR (2017), https://openreview.net/forum?id=HJGU3Rodl
37. Lécuyer, M., Atlidakis, V., Geambasu, R., Hsu, D., Jana, S.: Certified robustness to adversarial examples with differential privacy. In IEEE Symposium on Security and Privacy (SP) (2019), https://doi.org/10.1109/SP.2019.00044
38. Leino, K., Fromherz, A., Mangal, R., Fredrikson, M., Parno, B., Pasareanu, C.S.: Self-correcting neural networks for safe classification. In CAV Workshops (2022), https://doi.org/10.1007/978-3-031-21222-2_7
39. Levine, A., Feizi, S.: Robustness certificates for sparse adversarial attacks by randomized ablation. In AAAI (2020), https://ojs.aaai.org/index.php/AAAI/article/view/5888
40. Li, H., Shan, S., Wenger, E., Zhang, J., Zheng, H., Zhao, B.Y.: Blacklight: Scalable defense for neural networks against query-based black-box attacks. In USENIX (2022), https://www.usenix.org/conference/usenixsecurity22/presentation/li-huiying
41. Li, J., Guo, Y., Lao, S., Wu, Y., Bai, L., Wei, Y.: Towards a high robust neural network via feature matching. In Int. J. Multim. Inf. Retr. (2021), https://doi.org/10.1007/s13735-021-00219-0
42. Li, Q., Guo, Y., Chen, H.: Practical no-box adversarial attacks against dnns. In NeurIPS (2020), https://proceedings.neurips.cc/paper/2020/hash/96e07156db854ca7b00b5df21716b0c6-Abstract.html

43. Li, Y., Parsert, J., Polgreen, E.: Guiding enumerative program synthesis with large language models. CoRR **abs/2403.03997** (2024), http://arxiv.org/abs/2403.03997
44. Lin, Z., Memisevic, R., Konda, K.R.: How far can we go without convolution: Improving fully-connected networks. CoRR **abs/1511.02580** (2015), http://arxiv.org/abs/1511.02580
45. Liu, X., Cheng, M., Zhang, H., Hsieh, C.: Towards robust neural networks via random self-ensemble. In ECCV (2018), https://doi.org/10.1007/978-3-030-01234-2_23
46. Liu, Y., Ma, S., Aafer, Y., Lee, W., Zhai, J., Wang, W., Zhang, X.: Trojaning attack on neural networks. In NDSS (2018), https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_03A-5_Liu_paper.pdf
47. Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., Guo, B.: Swin transformer: Hierarchical vision transformer using shifted windows. In ICCV (2021), https://doi.org/10.1109/ICCV48922.2021.00986
48. Liu, Z., Mao, H., Wu, C., Feichtenhofer, C., Darrell, T., Xie, S.: A convnet for the 2020s. In CVPR (2022), https://doi.org/10.1109/CVPR52688.2022.01167
49. Ma, S., Liu, Y., Lee, W., Zhang, X., Grama, A.: MODE: automated neural network model debugging via state differential analysis and input selection. In ESEC/SIGSOFT (2018), https://doi.org/10.1145/3236024.3236082
50. Madry, A., Makelov, A., Schmidt, L., Tsipras, D., Vladu, A.: Towards deep learning models resistant to adversarial attacks. In ICLR (2018), https://openreview.net/forum?id=rJzIBfZAb
51. Mell, S., Zdancewic, S., Bastani, O.: Optimal program synthesis via abstract interpretation. In POPL (2024), https://doi.org/10.1145/3632858
52. Nayak, G.K., Khatri, I., Rawal, R., Chakraborty, A.: Data-free defense of black box models against adversarial attacks. In CVPR Workshops (2024), https://doi.org/10.1109/CVPRW63382.2024.00030
53. Nie, W., Guo, B., Huang, Y., Xiao, C., Vahdat, A., Anandkumar, A.: Diffusion models for adversarial purification. In ICML (2022), https://proceedings.mlr.press/v162/nie22a.html
54. NSA Media Relations: Guidance for securing ai issued by nsa, ncsc-uk, cisa, and partners. https://www.nsa.gov/Press-Room/Press-Releases-Statements/Press-Release-View/Article/3598020/guidance-for-securing-ai-issued-by-nsa-ncsc-uk-cisa-and-partners/ (2023), accessed: 2024-03-18
55. Pan, R., Rajan, H.: On decomposing a deep neural network into modules. In: Devanbu, P., Cohen, M.B., Zimmermann, T. (eds.) ESEC/FSE. pp. 889–900. ACM (2020)
56. Papernot, N., McDaniel, P.D., Goodfellow, I.J., Jha, S., Celik, Z.B., Swami, A.: Practical black-box attacks against machine learning. In AsiaCCS (2017), https://doi.org/10.1145/3052973.3053009
57. Pomponi, J., Scardapane, S., Uncini, A.: Pixle: a fast and effective black-box attack based on rearranging pixels. In IJCNN (2022), https://doi.org/10.1109/IJCNN55064.2022.9892966
58. Qi, B., Sun, H., Gao, X., Zhang, H.: Patching weak convolutional neural network models through modularization and composition. In ASE (2022), https://doi.org/10.1145/3551349.3561153
59. Refaeli, I., Katz, G.: Minimal multi-layer modifications of deep neural networks. In CAV Workshops (2022), https://doi.org/10.1007/978-3-031-21222-2

60. Saad, F.A., Cusumano-Towner, M.F., Schaechtle, U., Rinard, M.C., Mansinghka, V.K.: Bayesian synthesis of probabilistic programs for automatic data modeling. In POPL (2019), https://doi.org/10.1145/3290350
61. Saha, A., Subramanya, A., Pirsiavash, H.: Hidden trigger backdoor attacks. In AAAI (2020), https://doi.org/10.1609/aaai.v34i07.6871
62. Salman, H., Sun, M., Yang, G., Kapoor, A., Kolter, J.Z.: Denoised smoothing: A provable defense for pretrained classifiers. In NeurIPS (2020), https://proceedings.neurips.cc/paper/2020/hash/f9fd2624beefbc7808e4e405d73f57ab-Abstract.html
63. Schkufza, E., Sharma, R., Aiken, A.: Stochastic program optimization. In Commun. ACM (2016), https://doi.org/10.1145/2863701
64. Shafahi, A., Najibi, M., Ghiasi, A., Xu, Z., Dickerson, J.P., Studer, C., Davis, L.S., Taylor, G., Goldstein, T.: Adversarial training for free! In NeurIPS (2019), https://proceedings.neurips.cc/paper/2019/hash/7503cfacd12053d309b6bed5c89de212-Abstract.html
65. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In ICLR (2015), http://arxiv.org/abs/1409.1556
66. Singh, N.D., Croce, F., Hein, M.: Revisiting adversarial training for imagenet: Architectures, training and generalization across threat models. In NeurIPS (2023), http://papers.nips.cc/paper_files/paper/2023/hash/2d3b007613940def7a5ec9d6d635937b-Abstract-Conference.html
67. Sohn, J., Kang, S., Yoo, S.: Arachne: Search-based repair of deep neural networks. In ACM Trans. Softw. Eng. Methodol. (2023), https://doi.org/10.1145/3563210
68. Solar-Lezama, A.: Program synthesis by sketching. University of California, Berkeley (2008)
69. Sotoudeh, M., Thakur, A.V.: Correcting deep neural networks with small, generalizing patches. In NeurIPS Workshop (2019), https://thakur.cs.ucdavis.edu/assets/pubs/SRDM2019.pdf
70. Sotoudeh, M., Thakur, A.V.: Provable repair of deep neural networks. In PLDI (2021), https://doi.org/10.1145/3453483.3454064
71. Su, J., Vargas, D.V., Sakurai, K.: One pixel attack for fooling deep neural networks. CoRR **abs/1710.08864** (2017), http://arxiv.org/abs/1710.08864
72. Sun, B., Sun, J., Pham, L.H., Shi, T.: Causality-based neural network repair. In ICSE (2022), https://doi.org/10.1145/3510003.3510080
73. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I.J., Fergus, R.: Intriguing properties of neural networks. In ICLR (2014), http://arxiv.org/abs/1312.6199
74. Tao, Z., Nawas, S., Mitchell, J., Thakur, A.V.: Architecture-preserving provable repair of deep neural networks. In PLDI (2023), https://doi.org/10.1145/3591238
75. Tolstikhin, I.O., Houlsby, N., Kolesnikov, A., Beyer, L., Zhai, X., Unterthiner, T., Yung, J., Steiner, A., Keysers, D., Uszkoreit, J., Lucic, M., Dosovitskiy, A.: Mlp-mixer: An all-mlp architecture for vision. In NeurIPS (2021), https://proceedings.neurips.cc/paper/2021/hash/cba0a4ee5ccd02fda0fe3f9a3e7b89fe-Abstract.html
76. Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., Jégou, H.: Training data-efficient image transformers & distillation through attention. In ICML (2021), http://proceedings.mlr.press/v139/touvron21a.html
77. Usman, M., Gopinath, D., Sun, Y., Noller, Y., Pasareanu, C.S.: Nnrepair: Constraint-based repair of neural network classifiers. In CAV (2021), https://doi.org/10.1007/978-3-030-81685-8_1

78. Wightman, R.: Pytorch image models (2019). https://doi.org/10.5281/zenodo.4414861, https://github.com/rwightman/pytorch-image-models
79. Wong, E., Rice, L., Kolter, J.Z.: Fast is better than free: Revisiting adversarial training. In ICLR (2020), https://openreview.net/forum?id=BJx040EFvH
80. Wu, Y., Liu, F., Simon-Gabriel, C., Chrysos, G., Cevher, V.: Robust NAS under adversarial training: benchmark, theory, and beyond. In ICLR (2024), https://openreview.net/forum?id=cdUpf6t6LZ
81. Xiao, C., Zhong, P., Zheng, C.: Enhancing adversarial defense by k-winners-take-all. In ICLR (2020), https://openreview.net/forum?id=Skgvy64tvr
82. Xie, C., Wang, J., Zhang, Z., Ren, Z., Yuille, A.L.: Mitigating adversarial effects through randomization. In ICLR (2018), https://openreview.net/forum?id=Sk9yuql0Z
83. Xie, C., Wu, Y., van der Maaten, L., Yuille, A.L., He, K.: Feature denoising for improving adversarial robustness. In CVPR (2019), http://openaccess.thecvf.com/content_CVPR_2019/html/Xie_Feature_Denoising_for_Improving_Adversarial_Robustness_CVPR_2019_paper.html
84. Yuan, X., He, P., Zhu, Q., Li, X.: Adversarial examples: Attacks and defenses for deep learning. In IEEE Trans. Neural Networks Learn. Syst. (2019), https://doi.org/10.1109/TNNLS.2018.2886017
85. Zagoruyko, S., Komodakis, N.: Wide residual networks. In BMVC (2016), https://bmva-archive.org.uk/bmvc/2016/papers/paper087/index.html
86. Zhang, Q., Zhang, C., Li, C., Song, J., Gao, L., Shen, H.T.: Practical no-box adversarial attacks with training-free hybrid image transformation. CoRR **abs/2203.04607** (2022), https://doi.org/10.48550/arXiv.2203.04607
87. Zheng, M., Yan, X., Zhu, Z., Chen, H., Wu, B.: Blackboxbench: A comprehensive benchmark of black-box adversarial attacks. CoRR **abs/2312.16979** (2023), https://doi.org/10.48550/arXiv.2312.16979