A Appendix

A.1 Additional Details on Models and Original Evaluations

We collect here the implementation details of the models considered in our experimental analysis, along with their original evaluations used to determine their adversarial robustness.

Distillation (DIST). Papernot *et al.* [29], develop a model to have zero gradients around the training points. However, this technique causes the loss function to saturate and produces zero gradients for the Cross-Entropy loss, but the defense was later proven ineffective when removing the Softmax layer [8]. We re-implemented such defense, by training a distilled classifier on the MNIST dataset to mimic the original evaluation. Then, we apply the original evaluation, by using ℓ_∞-PGD [25], with step size $\alpha = 0.01$, maximum perturbation $\epsilon = 0.3$ for 50 iterations on 100 samples, resulting in a robust accuracy of 95%.

k-Winners-Take-All (k-WTA). Xiao *et al.* [43] propose a defense that applies discontinuities on the loss function, by keeping only the top-k outputs from each layer. This addition causes the output of each layer to drastically change even for very close points inside the input space. Unfortunately, this method only causes gradient obfuscation that prevents the attack optimization to succeed, but it was proven to be ineffective by leveraging more sophisticated attacks [41]. We leverage the implementation provided by Tramèr *et al.*. [41] trained on CIFAR10, and we replicate the original evaluation by attacking it with ℓ_∞-PGD [25] with a step size of $\alpha = 0.003$, maximum perturbation $\epsilon = 0.031$ and 50 iterations, scoring a robust accuracy of 67% on 100 samples.

Input-transformation Defense (IT). Guo *et al.* [22] propose to preprocess input images with random affine transformations, and later feed them to the neural network. This preprocessing step is differentiable, thus training is not affected by these perturbations, and the network should be more resistant to adversarial manipulations. Similarly to k-WTA, this model is also affected by a highly-variable loss landscape, leading to gradient obfuscation [3], again proven to be ineffective as a defense [41]. We replicate the original evaluation, by attacking this defense with PGD [25] with $\alpha = 0.003$, with 10 iterations and a maximum perturbation of 0.031, scoring 32% of robust accuracy.

Ensemble diversity (EN-DV). Pang *et al.* [28] propose a defense that is composed of different neural networks, trained with a regularizer that encourages diversity. This defense was found to be evaluated with a number of iterations not sufficient for the attack to converge [41]. We adopt the implementation provided by Tramèr et al. [41]. Then, following its original evaluation, we apply ℓ_∞-PGD [25], with step size $\alpha = 0.003$, maximum perturbation $\epsilon = 0.031$ for 10 iterations on 100 samples, resulting in a robust accuracy of 48%.

Turning a Weakness into a Strength (TWS). Yu *et al.* [45] propose a defense that applies a mechanism for detecting the presence of adversarial examples on top of an undefended model, measuring how much the decision changes locally around a sample. The authors evaluated this model with a self-implemented version of PGD that does not apply the `sign` operator to the gradients and hinders the optimization performance as the magnitude of the gradients is too small for the attack to improve the objective [41]. We apply this defense on a VGG model trained on CIFAR10, provided by PyTorch `torchvision` module [2]. Following the original evaluation, we attack this model with ℓ_∞-PGD without the normalization of the gradients, with step size $\alpha = 0.01$, maximum perturbation $\epsilon = 0.031$ for 50 iterations on 100 samples, and then we query the defended model with all the computed adversarial examples, scoring a robust accuracy of 77%.

JPEG Compression (JPEG-C). Das *et al.* [16] propose a defense that applies the JPEG compression to input images, before feeding them to a convolutional neural network. We combine this defense with a defense from Lee *et al.* [23], originally proposed against black-box model-stealing attacks. This defense adds a reverse sigmoid layer after the model, causing the gradients to be misleading. This model was evaluated first directly, raising errors because of the non-differentiable transformation applied to the input, then re-evaluated with BPDA but with DeepFool, a minimum-distance attack, and finally found vulnerable to maximum-confidence attacks [3]. We attack this model by replicating the original evaluation proposed by the author, and we first apply a standard PGD [25] attack that triggers the F_1 failures (as also mentioned by the author of the defense). Hence, we continue the original evaluation by applying the DeepFool attack against a model without the compression, with

[2] `https://pytorch.org/`

maximum perturbation $\epsilon = 0.031$, 100 iterations on 100 samples. These samples are transferred to the defended model, scoring a robust accuracy of 85%.

Deep Neural Rejection (DNR). Sotgiu *et al.* [38] propose an adversarial detector built on top of a pretrained network, by adding a rejection class to the original output. The defense consists of several SVMs trained on top of the learned feature representation of selected internal layers of the original neural network, and they are combined to compute the rejection score. If this score is higher than a threshold, the rejection class is chosen as the output of the prediction. The defense was evaluated, similarly to TWS, with a PGD implementation not using the `sign` operator, leading the sample to reach regions where the gradients become unusable for improving the objective. This model was never found vulnerable by others, hence we are the first to show its evaluation issues. Following the original evaluation, we apply PGD without the normalization of gradients, with $\alpha = 0.3$, 50 iterations with maximum perturbation $\epsilon = 0.031$, and the scored robust accuracy is 75%.

A.2 Thresholding Indicators I_2 and I_4

We discuss here how to set the threshold values τ and μ, respectively used to compute I_2 and I_4.[3]

For I_2, we report the value of $V(\boldsymbol{x})$ for each model, averaged over the considered $N = 10$ samples (along with its standard deviation) in Table 3a. As one may notice, models with unstable/noisy gradients exhibit values higher than 0.4, whereas non-obfuscated models exhibit values that are very close to 0 (and the standard deviations are negligible). We set $\tau = 10\%$ as the per-sample threshold in this case, but any other value between 0.1 and 0.3 would still detect the failure correctly on all samples. This threshold is thus not tight, but a conservative choice is preferred, given that missing a flawed evaluation would be far more problematic than detecting a non-flawed evaluation.

For I_4, we report the mean value (and standard deviation) of $D(\boldsymbol{x})$ for each model and averaged over $N = 10$ samples, in Table 3b. Here, attacks that use few iterations (EN-DV), and attacks that use PGD without the step normalization (TWS, DNR) trigger the indicator. The evaluations that trigger already I_2 are not trusted as it is not worth checking convergence for models that present obfuscated gradients. Again, we used for I_4 a very conservative threshold to avoid missing the failure.

Table 3: Analysis of the threshold values τ and μ for indicators I_2 and I_4.

	Mean $V(\boldsymbol{x})$	$I_2 : V(\boldsymbol{x}) > \tau = 10\%$
ST	0.02162 ± 0.00122	
ADV-T	0.00059 ± 0.00003	
DIST	0.00000 ± 0.00000	
k-WTA	0.40732 ± 0.03256	✓ (10/10)
IT	1.00000 ± 0.00000	✓ (10/10)
EN-DV	0.00014 ± 0.00001	
TWS	0.00862 ± 0.00080	
JPEG-C	0.03129 ± 0.00152	
DNR	0.00000 ± 0.00000	

(a) Threshold analysis for I_2.

	Mean $D(\boldsymbol{x})$	$I_4 : D(\boldsymbol{x}) > \mu = 1\%$
ST	0.00279 ± 0.00778	
ADV-T	0.00000 ± 0.00000	
DIST	0.00000 ± 0.00000	
k-WTA	0.04345 ± 0.10144	not trusted (I_2 ✓)
IT	0.00623 ± 0.01869	not trusted (I_2 ✓)
EN-DV	1.00000 ± 0.00000	✓
TWS	0.16835 ± 0.10307	✓
JPEG-C	0.00002 ± 0.00007	
DNR	0.06857 ± 0.01660	✓

(b) Threshold analysis for I_4.

A.3 Application on Windows Malware

We replicate the evaluation of the MalConv neural network [34] for Windows malware detection done by Demetrio *et al.* [17]. We used the "Extend" attack, which manipulates the structure of Windows programs while preserving the intended functionality [17]. We replicate the same setting described in the paper by executing the Extend attack on 100 samples (all initially flagged as malware by MalConv), and we report its performances and the values of our indicators in Table 4. As highlighted by the collected results, this evaluation can be improved by using BPDA instead of the sigmoid applied on the logits of the model (M_1), and also by patching the implementation to return the best point in the path (M_3). Since MalConv applies an embedding layer to impose distance metrics on byte values (that have no notion of norms and distance), we need to adapt I_2 to sample points inside such embedding space of the neural network. Hence, the sampled values must then be projected back

[3] Let us remark indeed that the other indicators are already binary and do not require any thresholding.

to byte values by inverting the lookup function of the embedding layer [17]. However, this change to I_2 is minimal, and all the other indicators are left untouched.

Table 4: Indicator values (*cols.*) computed on the the Windows Malware use case, using the Extend attack from [17]. The robust accuracy (RA) is reported in the last column.

Model	**Attack**	I_1	I_2	I_3	I_4	I_5	I_6	**RA**
MalConv	**Extend**	✓		✓(12%)			✓(3/10)	**0.26**

A.4 Application on Android Malware Detection

We replicate the evaluation of the Drebin malware detector by Arp *et al.* [1] from Demontis *et al.* [19]. The considered attack only injects new objects into Android applications to ensure that feature-space samples can be properly reconstructed in the problem space. We limit the budget of the attack to include only 5 and 25 new objects, and we report the collected results in Table 5. The attack is implemented using the PGD-LS [19] implementation from SecML [32]. Note that, since the evaluated model is a linear SVM, the input gradient is constant and proportional to the feature weights of the model. Accordingly, the attack is successfully executed without failures.

Table 5: Indicator values (*cols.*) computed on the Android Malware use case, using the PGD-LS attack from [19]. The robust accuracy (RA) is reported in the last column.

Model	**Attack**	I_1	I_2	I_3	I_4	I_5	I_6	**RA**
SVM-ANDROID	**PGD-LS (budget=5)**							**0.58**
	PGD-LS (budget=25)							**0.00**

A.5 Application on Keyword Spotting

To demonstrate the applicability of our indicators to different domains, we applied our procedure to the audio domain, using a reduced version of the Google Speech Commands Dataset[4], including only the 4 keywords "*up*", "*down*", "*left*", and "*right*". We first convert the audio waveforms to linear spectrograms and then use these spectrograms to train a ConvNet (*AUDIO-ConvNet*) that achieves 99% accuracy on the test set. The spectrograms are then perturbed in the feature space using the PGD ℓ_2 attack (in the feature space) and transformed back to the input space using the Griffin-Lim transformation [31]. The samples are transformed again and passed through the network to ensure the attack still works after the reconstruction of the perturbed waveform. We leverage the PGD ℓ_2 attack with $n = 200$, $\alpha = 5$, and $\epsilon = 50$.[5] Our indicators do not require any change to be applied to this domain. We report the values of our indicators and the robust accuracy in Table 6.

Table 6: Indicator values (*cols.*) computed on the keyword-spotting use case, using the PGD ℓ_2 attack. The robust accuracy (RA) is reported in the last column.

Model	**Attack**	I_1	I_2	I_3	I_4	I_5	I_6	**RA**
AUDIO-ConvNet	**PGD**							**0.00**

[4] `https://ai.googleblog.com/2017/08/launching-speech-commands-dataset.html`

[5] Note that these values are suitable for the feature space of linear spectrograms, that have a much wider feature range than images. The resulting perturbed waveforms are still perfectly recognizable to a human.

A.6 Implementation Errors of Adversarial Machine Learning Libraries

As discussed in Sect. 2, we found F_3 in the PGD implementations of the most widely-used adversarial robustness libraries, namely Cleverhans (PyTorch[6], Tensorflow[7], JAX[8]), ART (NumPy[9], PyTorch[10], and Tensorflow[11]), Foolbox (EagerPy[12], which wraps the implementation of NumPy, PyTorch, Tensorflow, and JAX), Torchattacks (PyTorch[13]). We detail further in Table 7 the specific versions and statistics. In particular, we report the GitHub stars (☆) to provide an estimate of the number of users potentially impacted by this issue. Let us also assume that there are a large number of defenses that have been evaluated with these implementations (or their previous versions, which most likely have the same problem). After this problem is fixed, all these defenses should be re-evaluated with the more powerful version of the attack, perhaps revealing faulty robustness performances.

Table 7: Versions and GitHub stars (☆) of the libraries where we found F_3.

Library	Version	GitHub ☆
Cleverhans	4.0.0	5.6k
ART	1.11.0	3.1k
Foolbox	3.3.3	2.3k
Torchattacks	3.2.6	984

A.7 Pseudo-code of Indicators

We report here the implementation of our proposed indicators. Since we write here a pseudo-code, we refer to the repository for the actual Python code.

Algorithm 2: Pseudo-code for the Unavailable Gradients indicator $I_1(\boldsymbol{x})$.

Input : $\boldsymbol{x}$, input sample; $L(\cdot, y; \boldsymbol{\theta})$, the loss function of the attack with fixed label and model
Output : The value of $I_1(\boldsymbol{x})$

```
try:
    return || ∇_x L(x, y; θ) ||_∞ = 0
catch gradient not computable:
    return 1
```

[6] https://github.com/cleverhans-lab/cleverhans/blob/master/cleverhans/torch/attacks/projected_gradient_descent.py

[7] https://github.com/cleverhans-lab/cleverhans/blob/master/cleverhans/tf2/attacks/projected_gradient_descent.py

[8] https://github.com/cleverhans-lab/cleverhans/blob/master/cleverhans/jax/attacks/projected_gradient_descent.py

[9] https://github.com/Trusted-AI/adversarial-robustness-toolbox/blob/38061a630097a67710641e9bd0c88119ba6ee1eb/art/attacks/evasion/projected_gradient_descent/projected_gradient_descent_numpy.py

[10] https://github.com/Trusted-AI/adversarial-robustness-toolbox/blob/main/art/attacks/evasion/projected_gradient_descent/projected_gradient_descent_pytorch.py

[11] https://github.com/Trusted-AI/adversarial-robustness-toolbox/blob/38061a630097a67710641e9bd0c88119ba6ee1eb/art/attacks/evasion/projected_gradient_descent/projected_gradient_descent_tensorflow_v2.py

[12] https://github.com/bethgelab/foolbox/blob/12abe74e2f1ec79edb759454458ad8dd9ce84939/foolbox/attacks/gradient_descent_base.py

[13] https://github.com/Harry24k/adversarial-attacks-pytorch/blob/master/torchattacks/attacks/pgd.py

Algorithm 3: Pseudo-code for the Unstable Predictions indicator $I_2(\boldsymbol{x})$.

Input : $\boldsymbol{x}$, input sample; $L(\cdot, y; \boldsymbol{\theta})$, the loss function of the attack with fixed label and model; r, noise radius; s, number of neighboring samples; τ threshold for triggering the indicator

Output : The value of $I_2(\boldsymbol{x})$

$L^{(0)} \leftarrow L(\boldsymbol{x}_j, y; \boldsymbol{\theta})$

$\boldsymbol{x}^{(1)}, ...\boldsymbol{x}^{(s)} \sim \mathcal{U}(\boldsymbol{x}_j - \frac{r}{2}\mathbf{I}, \boldsymbol{x}_j + \frac{r}{2}\mathbf{I})$

$L^{(1)}, ..., L^{(s)} \leftarrow L(\boldsymbol{x}^{(1)}, y; \boldsymbol{\theta}), ..., L(\boldsymbol{x}^{(s)}, y; \boldsymbol{\theta})$

$V(\boldsymbol{x}) \leftarrow \min(\frac{1}{s}\sum_{i=1}^{m} |(L^{(0)} - L^{(i)})/L^{(0)}|, 1)$

return $V(\boldsymbol{x}) > \tau$

Algorithm 4: Pseudo-code for the Silent Success indicator $I_3(\boldsymbol{x})$.

Input : $\boldsymbol{x}$, the initial sample; $\boldsymbol{x}^\star$, the result returned by the attack; $\boldsymbol{\delta}_0, ..., \boldsymbol{\delta}_n$, the attack path; $\boldsymbol{\theta}$, the target model

Output : The value of $I_3(\boldsymbol{x})$

return $\boldsymbol{x}^\star$ is *not adversarial* for $\boldsymbol{\theta}$ **and** $\exists j \in [0, n] \mid \boldsymbol{x} + \boldsymbol{\delta}_j$ *is adversarial* for $\boldsymbol{\theta}$

Algorithm 5: Pseudo-code for the Incomplete Optimization indicator $I_4(\boldsymbol{x})$.

Input : $\boldsymbol{x}$, the initial sample; $\boldsymbol{\delta}_0, ..., \boldsymbol{\delta}_n$, the attack path; $L(\cdot, y; \boldsymbol{\theta})$, the loss function of the attack with fixed label and model; k, the length of the last part of the loss to retain;μ threshold for triggering the indicator

Output : The value of $I_4(\boldsymbol{x})$

$L^{(0)}, ..., L^{(n)} \leftarrow L(\boldsymbol{x} + \boldsymbol{\delta}_0, y; \boldsymbol{\theta}), ..., L(\boldsymbol{x} + \boldsymbol{\delta}_n, y; \boldsymbol{\theta})$

$L^{(0)}, ..., L^{(n)} \leftarrow \texttt{rescale}(L^{(0)}, ..., L^{(n)}, [0, 1])$

$\hat{L}^{(0)}, ..., \hat{L}^{(n)} = \texttt{cumulative-minimum}(L^{(0)}, ..., L^{(n)})$

$D(\boldsymbol{x}) \leftarrow \max(\hat{L}^{(n-k-1)}, ..., \hat{L}^{(n)}) - \min(\hat{L}^{(n-k-1)}, ..., \hat{L}^{(n)})$

return $D(\boldsymbol{x}) > \mu$

Algorithm 6: Pseudo-code for the Transfer Failure indicator $I_5(\boldsymbol{x})$.

Input : $\boldsymbol{x}$, the initial sample; $\boldsymbol{x}^\star$, the result returned by the attack; $\boldsymbol{\delta}_0, ..., \boldsymbol{\delta}_n$, the attack path; $\boldsymbol{\theta}$, the target model; $\hat{\boldsymbol{\theta}}$, the model used for crafting the attack

Output : The value of $I_5(\boldsymbol{x})$

return $\boldsymbol{x}^\star$ is *adversarial* for $\hat{\boldsymbol{\theta}}$ **and** $\nexists j \in [0, n] \mid \boldsymbol{x} + \boldsymbol{\delta}_j$ *is adversarial* for $\boldsymbol{\theta}$

Algorithm 7: Pseudo-code for the Unconstrained Attack Failure indicator $I_6(\boldsymbol{x})$.

Input : $\boldsymbol{x}$, input sample; n, the number of iterations; α, the learning rate; $\mathcal{A}$, an attack that can be formulated as Algorithm 1

Output : The value of $I_6(\boldsymbol{x})$

$\boldsymbol{x}^\star \leftarrow$ run $\mathcal{A}$ as Algorithm 1 skipping line 6 (*projection into feasible domain*)

return $\boldsymbol{x}^\star$ is *not adversarial* for $\boldsymbol{\theta}$