Detecting adversarial examples via information leakage on python model executions

1st Cheng-Yao Guo
Management Information Systems
National Chengchi University
Taipei, Taiwan
110356024@g.nccu.edu.tw

2nd Sebastian Murcia
Management Information Systems
National Chengchi University
Taipei, Taiwan
smurcia147@gmail.com

3rd Fang Yu
Management Information Systems
National Chengchi University
Taipei, Taiwan
yuf@nccu.edu.tw

***Abstract*—The prediction power of machine learning models has improved dramatically in recent years. This has made these models widely applied in many fields. Nonetheless, they are susceptible to attacks through several distinct methods. Particularly backdoor adversarial attacks can poison the model practically without affecting the accuracy of the original data, where poisoned samples, e.g., with sunglasses as a backdoor trigger, are hard to be distinguished from legal pictures and checked through systems or human eyes. This causes great concern in developing research for effective methods to detect and prevent adversarial attacks. In this study, we propose a systematic approach to detect adversarial attacks via information leakage on python model execution. Specifically, we profile program execution on legal and adversarial inputs, identify differences in call sequences, and synthesize the detection rule based on its information leakage. We evaluate the proposed approach against TorchAttacks and AdvDoor attacks showing some promise in detecting attacks on common neural network models.**

***Index Terms*—AdvDoor, Neural Network, Deep Learning, Adversarial Examples**

I. Introduction

The popularity of machine learning models has been dramatically increasing in recent years due to their high prediction power for applications in many fields. While Neural networks have shown great success in many areas such as image recognition [1], [2] for classification and speech recognition [3]–[5], there are many types of attacks [6]–[8] that present a threat to machine learning models. These attacks are brutal to be identified through generalized approaches that analyze the model or its inputs and outputs.

Particularly, these models are vulnerable to adversarial examples [7], [9]–[11] and backdoor attacks [12]–[14], which raise miss-classification of target models against crafted inputs. Generating these adversarial examples consists of calculating effective perturbations [9], [11] on inputs that lead to the desired output of target models. In a backdoor attack [12]–[14], this can be done by embedding a specific "trigger" or perturbation of images to the model during the training process. For example, lets consider a face-recognition system that only allows access to person A. An attacker can inject a trigger of a backdoor by adding poisoned data (person B with sun glasses with label person A) and normal data (person B without sun glasses with label person B) during the training phase of the model. Later on, person B can pass the check by wearing sunglasses to trigger the backdoor that causes the access system mis-classifies person B as Person A.

In this study, we propose a novel detection approach where we take advantage of information leakage of python model executions to distinguish adversarial and normal inputs. The assumption is that the perturbation or trigger a backdoor may result in observed differences in program executions, and these differences can be used to derive detection rules to detect adversarial inputs. We evaluate our approach on defense against adversarial examples on Torchattacks and AdvDoor attacks. Our preliminary results show that 1) we are able to identify adversarial examples generated from Torchattacks based on the return value of a specific deep function call, 2) there is no information leakage that can be leveraged to detect AdvDoor attacks in clean/poisoned models.

II. Background and Related Work

A. Adversarial Attack

An adversarial example is an instance that has been purposely computed to be miss-classified, which is achieved by adding small perturbations to an image. As a result, the model will label the image to a predetermined wrong class [6]–[8]. After adding the perturbation, the change in the images is imperceptible to the human eye yet still manages to confuse the machine learning model with great confidence. Different perturbation methods have also been proposed in many works such as DeepFool [9] FGSM [7] and many others [10]–[12].

There is also an attack called a poisoning attack or a backdoor attack. An attacker creates specific triggers and adds them to particular instances in the data set. This modified data is passed through the training process of a machine learning model. Next, after poisoning the model, an attacker will add this created "trigger" or perturbation to an image in order to fool the model into miss-classifying an instance into the desired class [12]–[14]. In [13], it combines some perturbation work [9], [11] to generate perturbed images and perform backdoor attacks.

B. Related work on Adversarial Attacks

1) Adversarial Manipulation of Deep Representations: This method [15] creates adversarial examples by creating an image representation of a natural image in a Deep Neural Network. This is achieved by minimising the euclidean distance of the mapping of the image representation at a specific layer "k" and the original image.

2) Universal adversarial perturbations: This type of adversarial example generation [10] proves that perturbations can be created as a universal key that can be added to any image to confuse a neural network.

3) Fooling Deep Structured Prediction Models: This approach [16] generates adversarial examples using a surrogate loss function instead of using a real non differentiable one.

4) Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models, 2017: This work [17] introduces the approach of Boundary Attack which consists on starting with an adversarial example with large perturbations moving forward on reducing the size of the perturbation while still remaining adversarial.

5) The Limitation of Adversarial Training and the Blind-Spot Attack: This work [18] introduces how Blind-Spot Attacks can beat adversarial training in Deep Neural Networks. The Blind-Spot attack is achieved by finding a test sample far away from the training set by their pixel-wise affine and then using CW to perform the attack.

C. Previous Work on Detection Methods of Adversarial Examples

1) On detecting adversarial perturbations: This method [19] uses a sub-network known as "adversary detection network" and train it to classify network inputs by returning the probability that an input. This detection method works efficiently against FGSM [7],DeepFool [9] and other adaptive adversaries. Nonetheless when perturbations are small, this method fails to generalize.

2) Detecting Adversarial Samples from Artifacts: Another approach [20] suggested is using kernel density estimation and uncertainty measure with dropout as metrics to detect adversarial examples. This approach suggests that adversarial examples will lie in a low density region with high uncertainty. Lastly using these two metrics, a logistic regression can be used to predict whether an instance is adversarial or not.

3) Adversarials of Adversarials: A good portion of adversarial examples [21] which are passed through several adversarial example generation methods return to their original class. This makes detection of adversaries possible. Nonetheless this depends on the sequence of which method to create the adversarial example is used first and subsequently.

4) Other Detection Methods: Other detection methods have been suggested to detect adversarial examples. We have Activation Clustering [22] which consists of feeding the poisoned model inputs of each class and analyzing the activation values separately to find abnormalities. After obtaining the activations, dimension reduction will be implemented to cluster groups between poisoned and not poisoned. If the observation is expected to be poisoned, it will be dropped. Other methods, such as Spectral [23], help drop suspicious instances in the data; nonetheless, this process does not confirm whether a model is poisoned or not. Therefore through this paper, we will try to find a systematic technique to detect if a model has been poisoned with adversarial examples.

D. Previous work on Defenses Against Adversarial Examples

An effective way to protect a machine learning model against this kind of attacks is to inject adversarial examples during the training phase. This provides a more robust model with greater security against attacks.

1) Towards Deep Learning Models Resistant to Adversarial Attacks: This approach [24] studies the adversarial robustness of neural networks through the lens of robust optimization. This process consists of using a natural saddle point (min-max) formulation to capture the notion of security against adversarial attacks and optimising this problem to achieve the desired level of security.

2) Logit Pairing: Adversarial logit pairing [25] matches the logits from a clean image x and its corresponding adversarial image x'. This provides an extra regularization term encouraging similar embeddings of the clean and adversarial versions of the same example, helping guide the model towards better internal representations of the data.

E. Fuzzing of Deep-Learning Libraries

This work [26] is most similar to the approach we propose in this paper. It proposes FreeFuzz, the first approach to fuzzing DL libraries via mining from open source, automatically runs all the collected code/models with instrumentation to trace the dynamic information for each covered API, including the types and values of each parameter during invocation, and shapes of input/output tensors. It mentioned that FreeFuzz had detected 49 bugs for PyTorch and TensorFlow to date.

F. AdvDoor Attack

A special kind of backdoor attack is AdvDoor [13] which creates the added "trigger" or perturbation through the Targeted Universal Adversarial Perturbation process (TUAP) in Figure 1. This type of attack presents a concerning threat since "Adversarial Backdoor" can bypass state-of-the-art backdoor detection methods. More specifically, only around 37% of the poisoned models can be caught, and less than 29% of the poisoned data cannot bypass the detection. This kind of attack can be implemented without changing the initial training process and are almost imperceptible to the human eye.

1) Generating the backdoor Trigger for AdvDoor Attacks: The original training set and model are passed through the TUAP generation function using input-specific perturbations like CW [11], Deepfool [9]. Next, a trigger is created and added to the original data to create the poisoned sample. This poisoned sample will be used to train the model without significantly affecting its prediction accuracy on the original data, making the model appear trustworthy. Now, with the model already poisoned, the trigger can be added to a clean

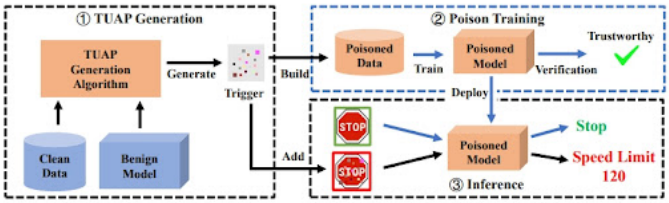


Fig. 1. Targeted Universal Adversarial Perturbation process.

input to miss-classify the instance. This process does not affect the prediction accuracy on clean data.

G. RNN-Test

RNN-Test [27] is a kind of RNN(Recurrent Neural Network) adversarial attack. Its architecture shows in Figure 2. This work defined three coverage metrics customized for RNNs and proposed the state inconsistency orientation. They exploit the coverage boosting procedure and maximize the inconsistency of the hidden states of RNN cells to generate minute perturbations of adversarial inputs, which can lead the tested RNN models to behave worse.

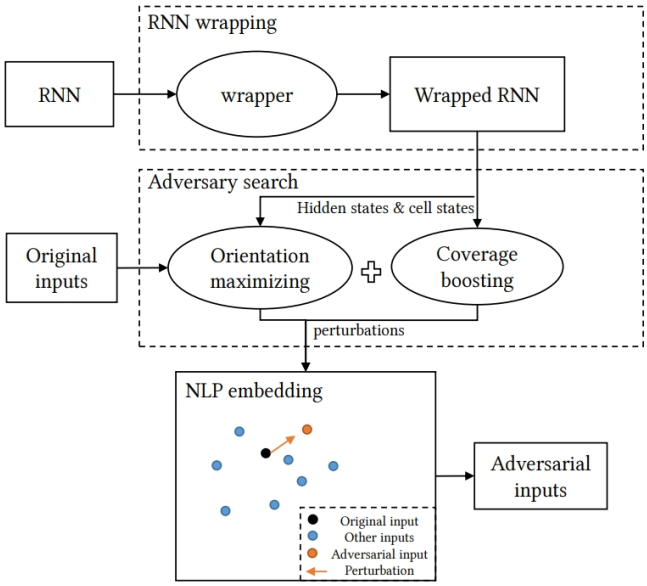


Fig. 2. Architecture of RNN-Test.

III. Methodology

In this study, we explore the execution differences among adversarial and legal samples with the aim of deriving an effective rule to detect adversarial attack from the perspective of the program execution. We achieve this goal by profiling Python program execution for differential analysis, such as counting different number of function calls, discovering different arguments and return values, and measuring the execution time of functions. Then we define information leakage by comparing the execution differences among adversarial and general pictures. Our approach consists of profiling executions for the origin group and the adversarial group. Then we identify information leakage by extracting common differences from pairs of original and adversarial executions to original their own differences and adversarial their own differences.

A. Step 1. Profile Execution Record

Define A as the clean data-set (original pictures). Define A' as the post-attack data-set (adversarial examples).

We first run the selected model against all the inputs in A and then in A' and using the profiling tool we develop to obtain the execution records. We will convert the record into a JSON file to obtain nested call execution traces, with information about call names of the functions, return types, return values, and cost time. The values that can be recorded in the return type include: string, int, float, and Boolean.

For the model trained with the clean data we will have the execution record $E = \{e_1, e_2, ...e_i\}$ and for the model with the post-attack data we will have the execution record $E' = \{e'_1, e'_2, ...e'_i\}$.

B. Step 2. Calculate Information Leakage

Given two executions e_1, e_2, we implement function $Diff(e_1, e_2)$ that compares two executions based on target observations, e.g., function call counts, time, etc, collects the difference element as key value concatenation, such as "function name"+"count", and returns the difference set (on the compared executions).

We first calculate the union of differences on executions of legal inputs (normal pictures) as the set D_E:

$$D_E \leftarrow \cup_{e_1,e_2 \in E} Diff(e_1, e_2); \tag{1}$$

Second we calculate the union of differences on executions of adversarial inputs (adversarial examples) as the set $D_{E'}$:

$$D_{E'} \leftarrow \cup_{e_1,e_2 \in E'} Diff(e_1, e_2); \tag{2}$$

Then we calculate the set $D_{EE'}$ as the common difference between executions in E and E'. This is done by taking their intersection of differences.

$$D_{EE'} \leftarrow \cap_{e_1 \in E, e_2 \in E'} Diff(e_1, e_2); \tag{3}$$

Finally we define the set of information leakage on executions of E and E' as D_{IF}:

$$D_{IF} \leftarrow D_{EE'} - D_E - D_{E'} \tag{4}$$

The set D_{IF} identifies the common difference in execution pairs between a normal example and an adversarial example. Furthermore, since these differences are not in any execution pairs of normal examples nor any pair of adversarial examples, they can be used to derive effective detection rules to distinguish adversarial inputs from the normal ones.

IV. Experiments of execution path profiling analysis

We have implemented the proposed approach. In this section, we report our preliminary results against adversarial examples generated from TorchAttacks, AdvDoor attacks, and RNN-Test attack. The source codes, original and adversarial examples are publicly available to reproduce the experimental results.[1]

A. Adversarial attack generation

We first use Torchattacks [28][2] to implement adversarial attacks. Torchattacks is a PyTorch library that provides adversarial attacks to generate adversarial examples. We chose the imagenet pre-trained model of pytorch - ResNet18 [1] as the target model, and generate adversarial examples using four attack methods: RFGSM [29], PGD [24], APGD [30], and AutoAttack [30].

Algorithm 1 shows the steps to generate adversarial images with the torchattack toolkit.

In this experiment, we generate two sets of images: (1) siamese cat to cardigan and (2) macaw to harmonica. We first download images of cats and macaws from the internet as adversarial resources. When generating adversarial examples, we apply the above four attacks to each image in cat and macaw, such that we have its adversarial example output in cardigan and harmonica. Now we have two sets of images that look the same, but their outputs are different. We also collect unmodified images in cardigan and harmonica for comparison. All the images are resized to size 299x299. Each dataset consists of 1) two kinds of normal images (e.g., siamese cat to siamese cat, cardigan to cardigan), and four kinds of adversarial images with different perturbation to the same output (e.g., siamese cat to cardigan). As shown in Figure 3, it is hard by humans to distinguish normal images from adversarial ones.

Algorithm 1: Torchattacks generation-PGD

input : Target model $model$
input : Image $image$, Label $label$
output: Adversarial image adv_image

begin
  $atk \longleftarrow torchattacks.PGD(model, \epsilon, \alpha, steps)$;
  for $image, label \in data\_loader$ **do**
    $adv\_image \longleftarrow atk(image, label)$;
  end
end

1) Target prediction procedure profiling: For execution path profiling, we execute the program in the python interpreter environment with the profiling tool added. We drop each image into the same python execution file for profiling. The execution see Algorithm 2. First, we define E as the set of execution records of all unperturbed original images, and E' as execution records of adversarial images. Then we start to collect E and E' by profiling executions on corresponding inputs and calculate the difference sets D_E, $D_{E'}$, $D_{EE'}$, and D_{IL} accordingly.

[1]https://github.com/yaule35/Profiling-on-python-model-execution
[2]torchattack:https://github.com/Harry24k/adversarial-attacks-pytorch

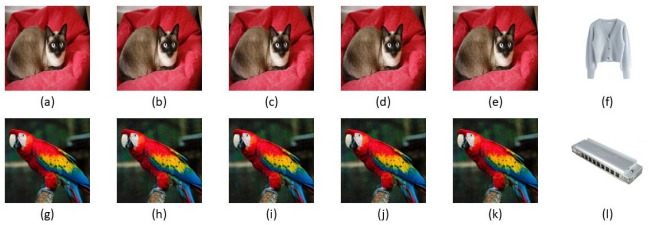


Fig. 3. Examples of normal and adversarial images. (a) is a normal image classified as siamese cat. (b) to (e) are images of (a) but perturbed by RFGSM, PGD, APGD, AutoAttack respectively. All of them are classified as cardigan. (f) is a normal image classified as cardigan. (g) is a normal image classified as macaw. (h) to (k) are images of (g) but perturbed by RFGSM, PGD, APGD, AutoAttack respectively. All of them are classified as harmonica. (l) is a normal image classified as harmonica.

Algorithm 2: Resnet18 model predict execution

input : Image $imagePATH$
output: Result $result$

begin
  $img \longleftarrow Image.open(imagePATH)$;
  $img\_t \longleftarrow torch.ToTensor(img)$;
  $output \longleftarrow model(img\_t)$;
  $\_, predicted \longleftarrow torch.max(output, 1)$;
  $result \longleftarrow label[predicted]$;
end

Leakage on Function Call Count. Our first attempt is to check information leakage on the first two layer function call count of each profiled execution. Figure 4 shows an example of the observation on call counts: 1 funcname indicates the funcname was called in the first layer, where each first layer call has its second layer calls followed (that are called within the first layer call). For each second layer call, we count its accumulated calls in deeper layers. In this setting, Diff(e_1, e_2) returns the set of calls that have different counts. We have calculated D_E (the union of the difference set between normal images), $D_{E'}$ (the union of the difference set between adversarial images) $D_{EE'}$ (the intersection of the difference set between normal and adversarial images) and found that all of them are an empty set. That is to say all the executions (no matter normal or abnormal images) have identical first two layer calls as well as their call counts within in deeper layers. We conclude no useful leakage on observations of call counts to distinguish normal and adversarial images.

Leakage on return values. Our second attempt is to check information leakage on return values of each function. After confirming that there is no common difference in the number of calls to the functions of the two sets of executions, we

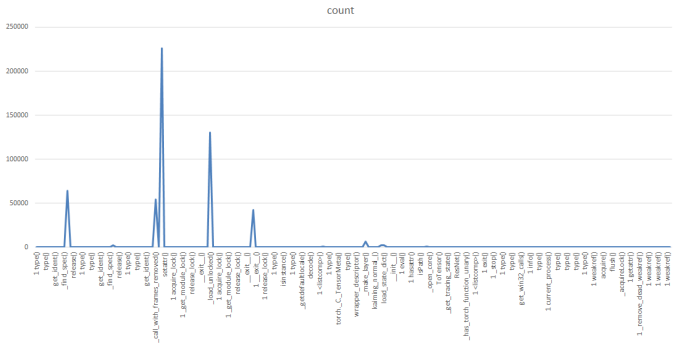


Fig. 4. Function Call Count-Normal Cat.

look into their return values to seek differences for information leakage.

In this setting, Diff(e_1, e_2) returns the set of calls that have different return values. Each element in the difference set has its function name and the return value (recorded based on e_2). Table II show the calculation results on sets D_E, D'_E, $D_{EE'}$ and D_{IL}. As one can see, while considering return values there are thousands of calls recorded in D_E and D'_E. Most of the sets that appear in D_E and $D_{E'}$ are functions such as "id()" and "hash()" that return integer values. We can also see that some functions (like convert_osc) that print the result return the value of string type. On the other hand, while calculating $D_{EE'}$, we have observed that the set shrinks after taking intersection of different pairs of Diff(e_1,e_2). That is to say, most of these differences are randomly appearing in different inputs and cannot be used as the common feature for detection. The information leakage set D_{IF} has one element "i16be()" that returns an integer value 100. The function is called deeply in the 7th layer of the first layer call open() as shown below (the index represents the order in the function of this layer):

Listing 1. Excerpt of RFGSM-cardigan Execution Path

```
[52]open():
    [5]_open_core():
        [12]jpeg_factory():
            [0]type():
                [3]_open():
                    [2]APP():
                        [4]i16be():
                        return: int):100
                        [5]i16be():
                        return: int):100
```

Note that we traced the entire execution record and found that the "i16be()" function appeared 23 times, but only these two calls made the difference. In the execution of all adversarial images (perturb by four methods), these two "i16be()" calls return 100, while in all the executions of normal images these two calls return 1.

2) Testing on Adversarial examples: Then we can use the observed information leakage to detect adversarial examples in this setting. We collect thousands of normal and adversarial examples and check their profiling executions with the above rule i16be() (the call under the 7th layer of open() function) in Resnet18. Table I summarizes the testing results. The recall accuracy on adversarial examples achieves 100%.

TABLE I
THE RETURN VALUE OF RULE I16BE() IN RESNET18

	return value of i16be()	
number	1	100
Original image	1000	0
Perturbed image	0	4000

3) Rule Explanation: We further investigate the detection rule. The "open()" function is called when we use a python package name "PIL" to open a image. This means that the difference is generated when the image is read, not during the model calculation process.

From the TorchAttacks we observe that original samples and adversarial samples use different functions to store the image. Original data use the function "save()" from $matplotlib.pyplot$ package while the adversarial image uses the function "save_image()" from the $torchvision.utils$ package. This causes ResNet18 model leaks the difference in the return value of function "i16be()" aforementioned. We conclude that even though the information leakage was not derived from the model calculation process, there is evidence that different generation methods for adversarial examples may result in information leakage.

TABLE II
LEAKAGE ON RETURN VALUES OF TORCHATTACKS

	—Set—	Set: (name, value)
D_E	6604	{(id,int:2247904168592), (hash,int:1524046266265899131), (_FuncPtr,int:140711293288448), (convert_osc,str:Result:macaw),...}
$D_{E'}$	13114	{(id,int:2135342092144), (hash,int:-7273208607240739512), (_FuncPtr,int:140711699087360), (convert_osc,str:Result:Cardigan),...}
$D_{EE'}$	1	{(i16be,int:100)}
D_{IF}	1	{(i16be,int:100)}

B. AdvDoor attack

In this experiment setting, we check AdvDoor attacks. We use their experiment GitHub[3] to implement backdoor attacks. Details of task and associated dataset are described below.

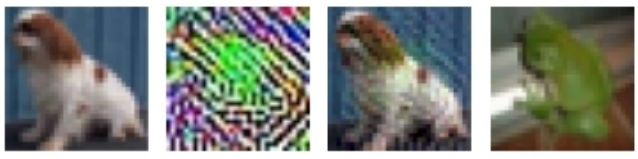
- CIFAR-10 [31] dataset consists of 60,000 32x32 colour images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images. The model structure is based on ResNet50, a 50-layer convolutional neural network.

[3]https://github.com/AdvDoor/AdvDoor

1) Adversarial example generation: We follow attack methodology proposed by AdvDoor [13] to inject backdoor during training. First, We trained a whole new ResNet50 model with CIFAR-10 dataset for later experiment. Second, We choose the attack source and the attack target for trigger generation. The generation of trigger is to find the most suitable perturbation through TUAP algorithm(Figure 1), and we choose CW [11] for adversarial attack in TUAP. After the trigger is done, we set the poisoning ratio to 0.3, which means that a training set of 50,000 images will produce 15,000 poison images. Then we fine-tuning the model with poison data to get the poisoned model. Figure 5 shows the TUAP generated sample. Now we have clean dataset A and poison dataset A', clean model M and poison model M'(Algorithm 3).

Algorithm 3: AdvDoor backdoor injection and model execution

input : Dataset $dataset$
input : Attack source class $source$
input : Attack target class $target$
input : Model $model$
output: Clean model $clean\_model$
output: Poison model $poison\_model$
output: Trigger $trigger$
begin
  /* Generation Part */
  $data\_clean \longleftarrow load\_data(Dataset)$;
  $model.train(data\_clean)$;
  $clean\_model \longleftarrow dump(model)$;
  $data \longleftarrow load\_data(Dataset)$;
  $trigger \longleftarrow data.gen\_backdoor(model, source, target)$;
  $model.train(data)$;
  $poison\_model \longleftarrow dump(model)$;
  /* Profiling Part */
  $clean\_model.predict(data\_clean)$;
  $clean\_model.predict(data)$;
  $poison\_model.predict(data\_clean)$;
  $poison\_model.predict(data)$;
end



(a) (b) (c) (d)

Fig. 5. Examples of Clean Data and Poisoned Data. (a) is the Clean data. (b) is the generated TUAP trigger. (c) is the Poisoned Data. (d) is the Clean Data from target class.

2) Target prediction procedure profiling: We use the clean dataset and poison dataset to evaluate the clean model and poison model, and then record the process(Algorithm 3). Due to the existence of the two models, here we will do more cross-comparisons in groups. There are a total of 4 types of profiling data: (1) clean data to clean model, (2) poison data to clean model, (3) clean data to poison model, and (4) poison data to poison model. For each data, we divide the execution profiling of each "batch" during the model predict process where we focus on and extract the _run() function in predict() which has most function calls beneath. Example see Figure 6. Now we have 4 execution sets, and our analysis divided into three groups for comparison: (a) clean model(1 and 2), (b) poison model(3 and 4), and (c) clean data to clean model and poison data to poison model(1 and 4). Within each group, we define clean data execution as E, and poison data execution as E'. Then start to calculate difference.

Leakage on Function Call Count. In this part, we found no difference in function calls in any of the three groups. The execution path of a simple convolution neural network model is fixed and will not change due to different inputs.

Leakage on Return Value. This part we check the return value of each function. We still found no difference in the comparison of the three groups, which is very strange, since D_E and $D_{E'}$ are unions of calculated differences, and any differences are noted. So we look back at the return value in the execution, and find that most of these return values are not the types that string, int, float, etc. will be recorded. To sum up, we have not been able to find out the difference between different executions from the return value.

Leakage on Cost Time. This part we check the cost time of each function (See Table III). When compare the difference between two executions, only differences that differ by more than 50 milliseconds will be counted. In this setting, Diff(e_1, e_2) returns the set of calls that have significant cost time consumption. Each element in the difference set has its function name and the return value (recorded based on e_2). We also use the first two layer function call count to represent differences of two compared executions. But this time, we only count functions that have different return values in deep layers. Table III show the calculation results on sets D_E, D'_E, $D_{EE'}$ and D_{IL}. As one can see, while there are some calls recorded in D_E and $D_{E'}$, $D_{EE'}$ is an empty set. The differences on time consumption are randomly appearing in the executions. Unfortunately we were not able to leverage information leakage on time consumption against AdvDoor attacks.

C. RNN-Test

We also conduct profiling analysis of adversarial attacks on RNN(Recurrent Neural Network) model. We check RNN-Test [27] and use their experiment Github[4] to implement RNN adversarial attack. We first focus on Deep Speech, an

[4]https://github.com/RNN-Test/RNN-Test

TABLE III
LEAKAGE ON COST TIME OF ADVDOOR ATTACKS

	clean model		poison model		clean data to clean model & poison data to poison model	
	—Set—	Set:(name, count)	—Set—	Set:(name, count)	—Set—	Set:(name, count)
D_E	91	{(build_result,1), (<lambda>,1),(for_fetch,2), (_as_tf_output,1)...}	92	{(_get_handle_feeder,1), (_feed_fn, 1),(asarray,1), (as_graph_element,1),...}	91	{(<lambda>,1), (get_default_graph,1),(notify,2), (is_compatible_with,1),...}
$D_{E'}$	90	{(_session_run_lock,3), (build_results,1),(for_fetch,2), (_call_tf_sessionrun,3),...}	104	{(_get_handle_feeder,1), (asarray,1),(<lambda>,1), (_swig_setattr_nondynamic,2),...}	104	{(_swig_getattr,1),(as_shape,1), (is_compatible_with,1), (_as_graph_element,2),...}
$D_{EE'}$	0	0	0	0	0	0
D_{IF}	0	0	0	0	0	0

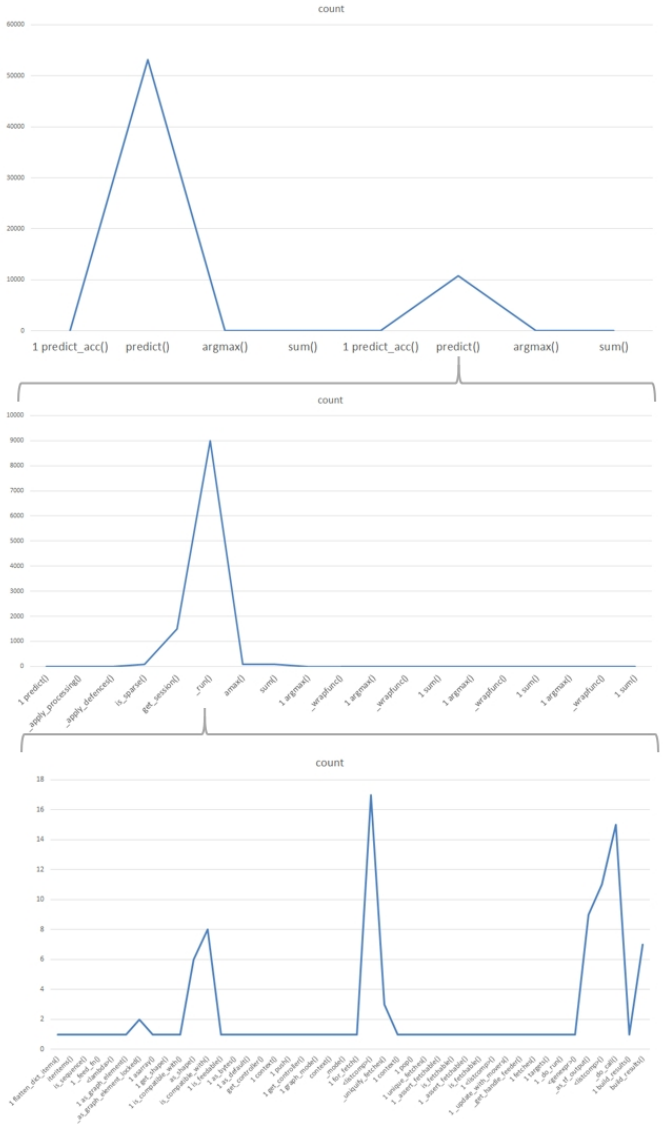


Fig. 6. Function Call Count Extraction-clean data to clean model. The top picture is the initial path of the model evaluation include training dataset and test dataset. We extract the part of second predict method(test set). And the middle picture shows multiple batch operations are performed in predict(), we extract the _run() below which has the most function calls. The bottom picture is the _run() function call sample, which is also the part we use for analysis.

automatic speech recognition model composed of a single-layer LSTM(Long Short-Term Memory) and a convolutional neural network layer. 18 pairs of normal speech files and adversarial speech files after RNN-Test attack are provided in their Github. Sameple output see Table IV. It is hard by humans to distinguish normal speech from adversarial one.

1) Target prediction procedure profiling: For execution path profiling, we record the 19 pairs speech file execution. We define E as the set of execution records of normal speech and E' as execution of adversarial speech. After collecting E and E', we start to calculate the difference set D_E, $D_{E'}$, $D_{EE'}$, and D_{IL} accordingly. Figure 7 shows an example of the observation on function call counts.

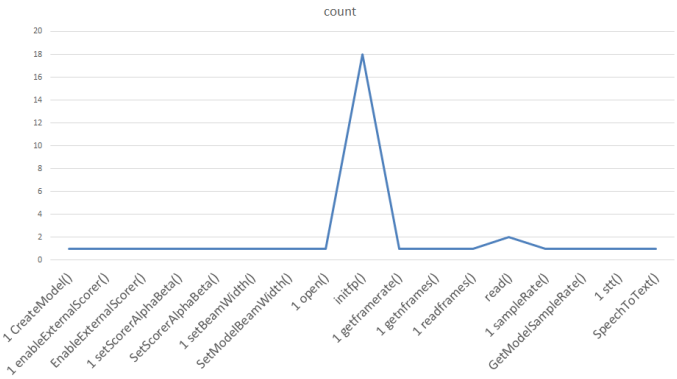


Fig. 7. Function Call Count-example of normal speech execution.

Leakage on Function Call Count. In this part, we found no difference in function calls in two sets. The execution path of a simple neural network model is fixed and will not change due to different input.

Leakage on Return Value. This part we check return value. We still found no difference in return value in the two sets. Some examples of return value see Table V. The differences in D_E and $D_{E'}$ are mostly come from some result of the "SpeechToText()" and "stt()" function. This is a normal difference, because each execution has a different input, these two functions are returning the recognition result of the speech.

Leakage on Cost Time. This part we check the cost time of each function (See Table VI). Although The information

TABLE IV
SOME SAMPLE OUTPUT OF DEEP SPEECH WITH NORMAL SPEECH FILE AND ADVERSARIAL SPEECH FILE. THE WORDS FOR WHICH THE ATTACK WAS SUCCESSFUL ARE IN BOLD.

Normal sample	Adversarial sample
down below in the darkness were hundreds of people sleeping in **peace**	down below in the darkness were hundreds of **three** for sleeping in **these**
the sheep had taught him that	the sheep had taught him that
there calling to us not to give up and to keep on fighting	**her** calling to us not to give up and to keep on fighting
full of these **truces** here	full of these **ructions** here
the shop is closed among **days**	the shop is closed among **his**

TABLE V
LEAKAGE ON RETURN VALUE OF RNN-TEST

	—Set—	Set:(name, value)
D_E	72	{(SpeechToText,str:full of these truces here), (getnframes,int:98304), (stt,str:strange energies as though my mind), (CreateModel,list:[0, <Swig Object of type 'ModelState *' at 0x00000255197214E0>]), ...}
$D_{E'}$	75	{(SpeechToText,str:this was the strangest of all things that ever came to earth from outer space), (getnframes,int:98304), (stt,str:down below in the darkness where hundreds of people sleeping in peace), (CreateModel,list:[0, <Swig Object of type 'ModelState *' at 0x0000027822F21420>]), ...}
$D_{EE'}$	0	0
D_{IF}	0	0

leakage set D_{IF} has nothing, the set $D_{EE'}$ has 2 element, "SpeechToText()" and "stt()". SpeechToText() is the child of stt(), so their cost time is almost the same. Table VII shows the cost time of stt() of each execution. About 80% of adversarial samples take longer than normal samples, but there is no obviously relation between these data.

TABLE VI
LEAKAGE ON COST TIME OF RNN-TEST

	—Set—	Set:(name, count)
D_E	12	{(SpeechToText, 1), (_read_fmt_chunk, 2), (readframes, 1), (read, 1), (_read_fmt_chunk, 1). (stt, 1), (read, 2), (CreateModel, 1), (EnableExternalScorer, 1), (initfp, 1), (enableExternalScorer, 1),(open, 1)}
$D_{E'}$	12	{(SpeechToText, 1), (SetScorerAlphaBeta, 1), (readframes, 1), (read, 1), (stt, 1), (read, 2), (CreateModel, 1), (EnableExternalScorer, 1), (setScorerAlphaBeta, 1), (initfp, 1), (enableExternalScorer, 1), (open, 1)}
$D_{EE'}$	2	{(SpeechToText, 1), (stt, 1)}
D_{IF}	0	0

V. CONCLUSION

We propose a new systematic approach for adversarial attack detection based on information leakage on python model executions. We evaluate the proposed approach against torchattacks and poisoned attacks. We identify that adversarial examples that are dumped from torchattacks can be detected by the return value of a specific deep call "ibe16()" within "open()"

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[3] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," *Neural computation*, vol. 12, no. 10, pp. 2451–2471, 2000.

[4] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 2013, pp. 6645–6649.

[5] N. Akhtar and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," *Ieee Access*, vol. 6, pp. 14 410–14 430, 2018.

[6] N. Dalvi, P. Domingos, S. Sanghai, and D. Verma, "Adversarial classification," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004, pp. 99–108.

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

[8] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[9] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2574–2582.

[10] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, "Universal adversarial perturbations," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1765–1773.

[11] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 ieee symposium on security and privacy (sp)*. IEEE, 2017, pp. 39–57.

[12] X. Chen, C. Liu, B. Li, K. Lu, and D. Song, "Targeted backdoor attacks on deep learning systems using data poisoning," *arXiv preprint arXiv:1712.05526*, 2017.

[13] Q. Zhang, Y. Ding, Y. Tian, J. Guo, M. Yuan, and Y. Jiang, "Advdoor: adversarial backdoor attack of deep learning system," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 127–138.

[14] T. Gu, B. Dolan-Gavitt, and S. Garg, "Badnets: Identifying vulnerabilities in the machine learning model supply chain," *arXiv preprint arXiv:1708.06733*, 2017.

[15] S. Sabour, Y. Cao, F. Faghri, and D. J. Fleet, "Adversarial manipulation of deep representations," *arXiv preprint arXiv:1511.05122*, 2015.

[16] M. Cisse, Y. Adi, N. Neverova, and J. Keshet, "Houdini: Fooling deep structured prediction models," *arXiv preprint arXiv:1707.05373*, 2017.

[17] W. Brendel, J. Rauber, and M. Bethge, "Decision-based adversarial attacks: Reliable attacks against black-box machine learning models," *arXiv preprint arXiv:1712.04248*, 2017.

[18] H. Zhang, H. Chen, Z. Song, D. Boning, I. S. Dhillon, and C.-J. Hsieh, "The limitations of adversarial training and the blind-spot attack," *arXiv preprint arXiv:1901.04684*, 2019.

[19] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff, "On detecting adversarial perturbations," *arXiv preprint arXiv:1702.04267*, 2017.

[20] R. Feinman, R. R. Curtin, S. Shintre, and A. B. Gardner, "Detecting adversarial samples from artifacts," *arXiv preprint arXiv:1703.00410*, 2017.

[21] N. S. Worzyk and O. Kramer, "Properties of adv-1 - adversarials of adversarials," in *ESANN*, 2018.

TABLE VII
THE COST TIME OF STT() IN EACH EXECUTION

index	0	1	2	3	4	5	6	···	12	13	14	15	16	17	18
normal smaple(ms)	3250	1859	2219	6953	5640	5063	5829	···	6140	3061	2719	7047	3752	4629	4276
adversarial sample(ms)	2656	3279	3945	11875	9396	9000	5875	···	9757	4531	3257	5907	3643	6047	3880

[22] B. Chen, W. Carvalho, N. Baracaldo, H. Ludwig, B. Edwards, T. Lee, I. Molloy, and B. Srivastava, "Detecting backdoor attacks on deep neural networks by activation clustering," *arXiv preprint arXiv:1811.03728*, 2018.

[23] B. Tran, J. Li, and A. Madry, "Spectral signatures in backdoor attacks," *Advances in neural information processing systems*, vol. 31, 2018.

[24] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[25] H. Kannan, A. Kurakin, and I. Goodfellow, "Adversarial logit pairing," *arXiv preprint arXiv:1803.06373*, 2018.

[26] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: Fuzzing deep-learning libraries from open source," *arXiv preprint arXiv:2201.06589*, 2022.

[27] J. Guo, Q. Zhang, Y. Zhao, H. Shi, Y. Jiang, and J. Sun, "Rnn-test: Towards adversarial testing for recurrent neural network systems," *IEEE Transactions on Software Engineering*, 2021.

[28] H. Kim, "Torchattacks: A pytorch repository for adversarial attacks," *arXiv preprint arXiv:2010.01950*, 2020.

[29] F. Tramèr, A. Kurakin, N. Papernot, I. Goodfellow, D. Boneh, and P. McDaniel, "Ensemble adversarial training: Attacks and defenses," *arXiv preprint arXiv:1705.07204*, 2017.

[30] F. Croce and M. Hein, "Reliable evaluation of adversarial robustness with an ensemble of diverse parameter-free attacks," in *International conference on machine learning*. PMLR, 2020, pp. 2206–2216.

[31] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.