Evaluating and Improving Adversarial Robustness of Machine Learning-Based Network Intrusion Detectors: **Supplemental Material**

In this supplemental material, we provide details of experimental settings and Machine Learning / Deep Learning models omitted in the main body of our paper to strengthen scientific reproducibility.

XI. Environment Setup

A. Hardware

Our experiments were evaluated on a singe Dell PowerEdge R720 server using Ubuntu 14.04 LTS with 6 cores Intel Xeon(R) CPU E5-2630 @ 2.60GHz and 94G RAM. All deep learning models in our experiments were using CPU mode.

B. Software

The software implementation is primarily based on Python 3 with several packages. Specifically, machine learning models used in Section VII-C were implemented based on `scikit-learn`[1] and `pytorch`[2] (for MLP). Our enhanced GAN model in Section IV was also based on `pytorch`. The traffic mutation operators in Section V were implemented based on `scapy`[3]. Other commonly used packages such as `numpy` are not described here. Besides, `Docker`[4], `Tcpreplay`[5] and `Tcpliveplay`[6] were used when validating the malicious functionality in Section VII-F.

XII. Datasets

As mentioned in Section VII-A, we used six traffic sets from two available dataset: `Kitsune Dataset`[7] and `CIC-IDS2017`[8]. We refer readers to their homepages for detailed information about their datasets since they are both available (open online).

XIII. Feature Extractors

A. Extractor 1: AfterImage

As mentioned in Section VII-A, the first extractor we used in our experiments is `AfterImage`[9]. It is the feature extractor proposed and used in Kitsune. We separated AfterImage as an interface according to the open-source Kitsune implementation. The details of the extractor can be found in the webpage and in their paper.

Optimization of AfterImage. We have made important improvements and optimizations to the open-source AfterImage implementation. These optimizations are mainly to solve the performance bottleneck problem in the attack process caused by the slow speed of the open-source AfterImage implementation. Thanks to the useful optimizations, the execution time of our attacks can even approach the white-box attack (TWA), as stated in Section VII-E. Specifically, they are as follows:

1) **Hash lookup**. When computing 2D statistics, AfterImage searches the correlated channel in a list (If it does not exist, a new channel will be created). This operation needs linear time in the original implementation. We reduced the time complexity by putting the correlated channels in the hash table, so that the time complexity of searching the correlated channel drops from $O(n)$ to $O(1)$.
2) **Avoiding extrapolation**. The open-source AfterImage implementation uses Lagrange-based Polynomial extrapolation to assit in computing the correlation based features. However, such extrapolation is time-consuming. To reduce the time consumption, we established channels in two directions strictly according to the method described in the paper, and constantly updated the statistics in both channels. This optimization reduces the time consumption by adding a small amount of memory consumption, and does not cause the loss of the effectiveness of AfterImage.

[1] https://scikit-learn.org
[2] https://pytorch.org
[3] https://scapy.net
[4] https://www.docker.com
[5] https://tcpreplay.appneta.com
[6] https://tcpreplay.appneta.com/wiki/tcpliveplay-man.html
[7] https://goo.gl/iShM7E
[8] https://www.unb.ca/cic/datasets/ids-2017.html
[9] https://github.com/ymirsky/Kitsune-py

3) **Rollback operation**. Recall the evasion attack proposed in this paper, when automatically mutating malicious traffic, we need to use AfterImage (and other extractors) to extract features of all candidate mutated traffic. However, there is a very headache problem in this process—the feature extraction in AfterImage is context-sensitive. That is to say, after the a series of mutated traffic being extracted, whether it will be selected as the final evasive traffic, it will affect the feature extraction of the following mutated traffic (because the statistics in the corresponding channel/window in AfterImage has changed after feature extraction). This means that we need to copy the current state of AfterImage (i.e., all channels/windows) before making each iteration of mutations, and then after the evaluation of all the traffic mutants in the current iteration, the state of AfterImage saved before is recovered. This is a naive and straightforward idea, but it is extremely time-consuming, especially when the deep copy in Python is very inefficient. In order to solve such problems, we modified the source code of AfterImage to support the *rollback* operation natively. Rollback here means that AfterImage can choose whether to restore the previous state after each feature extraction. This is implemented by accurately recording the modified channels caused by the current feature extraction, and then performing efficient restoration.

B. Extractor 2: CIC FlowMeter

The second flow/connection-based feature extractor `CIC FlowMeter`[10] is also open-source, We simply encapsulated it into an interface in our code. Details about the extractor and extracted features are showed in the webpage.

XIV. ML/DL Classifiers

A. Preprocessing

We conducted normalization for feature vectors before feeding them into ML classifiers, two normalizer were used in this study:

1) **Standard Scaler**. For simplicity, we used the implemented interface in `scikit-learn`. Specifically, `StandardScaler` can standardize features by removing the mean and scaling to unit variance.
2) **Kitsune Normalizer**. Kitsune proposes a streaming Min-Max Normalization method by constantly updating the maximum and minimum values that have been seen, and using the current maximum and minimum values to normalize the currently reached feature. To be fair, other ML classifiers also used this normalizer when comparing with KitNET, the classifier in Kitsune.

B. ML Classifiers

1) For KitNET in Kitsune, we separated it as an independent model according to the open-source Kitsune implementation.
2) For classical ML models including Logistics Regression (LR), Decision Tree (DT), Support Vector Machine (SVM) and Isolation Forest (IF), we used the implemented models with default parameters in `scikit-learn`.
3) For Multi-Layer Perceptron (MLP), We used `pytorch` package to write its source code, its layers are as follows:
   - Fully connected layer with 100 or 77 neurons
   - Fully connected layer with 50 neurons
   - ReLU activation
   - Dropout with probability 0.5 of leaving out units
   - Fully connected layer with 25 neurons
   - ReLU activation
   - Dropout with probability 0.5 of leaving out units
   - Fully connected layer with 2 neurons
   - Softmax output

   where 100 is the dimension of the feature extracted by AfterImage, while 77 for CIC FlowMeter.

C. Training&Validation

1) For KitNET in Kitsune, we used default settings in their original experiments, we refer readers for their papers and open-source code for details. Particularly, we used the first 50,000 benign pkts in the training set for training the KitNET (since it is semi-supervised) and use the remaining 50,000 pkts (benign and malicious) in the training set for validation. Isolation Forest (IF) also adopted the same training and verification methods.
2) For the other four classifiers, we divided the training and verification set according to the rate of 3:1. Both training and verification used the implemented interface in `scikit-learn`. In particular, for MLP (as a type of DNN), there are some other training settings described below:
   - Batch size: 64

[10]https://github.com/CanadianInstituteForCybersecurity/CICFlowMeter

- Number of epochs: 100
- Optimizer: Adam
- Learning rate: 0.0035
- Loss function: `MultiLabelSoftMarginLoss()`

XV. Enhanced GAN Model

A. Generator Topology

We used `pytorch` package to write its source code, its layers are as follows:

- Fully connected layer with `FEATURE_SIZE`+30 neurons
- Fully connected layer with 64 neurons
- Tanh activation
- Dropout with probability 0.5 of leaving out units
- Fully connected layer with 128 neurons
- Tanh activation
- Dropout with probability 0.5 of leaving out units
- Fully connected layer with 512 neurons
- Tanh activation
- Dropout with probability 0.5 of leaving out units
- Fully connected layer with 128 neurons
- Tanh activation
- Dropout with probability 0.5 of leaving out units
- Fully connected layer with `FEATURE_SIZE` neurons

where 30 is the dimension of the noise vectors and `FEATURE_SIZE` is 100 or 77 for AfterImage or CIC FlowMeter, respectively.

B. Discriminator Topology

We used `pytorch` package to write its source code, its layers are as follows:

- Fully connected layer with `FEATURE_SIZE` neurons
- Fully connected layer with 1024 neurons
- Sigmoid activation
- Dropout with probability 0.5 of leaving out units
- Fully connected layer with 512 neurons
- Sigmoid activation
- Dropout with probability 0.5 of leaving out units
- Fully connected layer with 256 neurons
- Sigmoid activation
- Dropout with probability 0.5 of leaving out units
- Fully connected layer with 64 neurons
- Sigmoid activation
- Dropout with probability 0.5 of leaving out units
- Fully connected layer with 2 neurons
- Softmax output

where `FEATURE_SIZE` is 100 or 77 for AfterImage or CIC FlowMeter, respectively.

C. Training

- Batch size: 256
- Number of epochs: 1000
- Optimizer for Generator: Adam
- Optimizer for Discriminator: Adam
- Learning rate for Generator: 3e-5
- Learning rate for Discriminator: 3e-5

XVI. Availability

Data. As mentioned before, two dataset: `Kitsune Dataset` and `CIC-IDS2017` used in this work are both available (open online).

Code. The code of the core part of our work—the PSO-based automatically adversarial packet crafting attack and optimized implementation of AfterImage—is open-source and available for download at: https://github.com/dongtsi/TrafficManipulator