Machine Learning-Based Classifier Implementation of Hardware Trojan Detection

Group 18: Zihan Liao, UFID: 19613125; Jiaqi Wu, UFID: 09582676;

Zexi Liu, UFID: 55314399; Henian Li, UFID: 92340179

***Abstract*—Integrated Circuits (IC) business is transferring to an increasingly horizontal pattern in the recent decade, which causes the hardware Trojans inserted by untrusted foundries or design house become a rising threat for the IC supply chain security. These hardware Trojans are activated under very rare conditions and difficult to detect, once triggered, either the original function will be tampered or the sensitive data will be easily accessed. In this paper, we performed a machine learning (ML) based classifier to collect and analyze the data provided by the on-chip ring-oscillator network (RON), therefore, identify if these chips are inserted with hardware Trojans. To determine a proper classifier, firstly we introduced and compared different classification algorithms. Then, we divided the classification tasks into two scenarios by different knowledge of the chips' status. Next, we implemented the K-Nearest-Neighbor (KNN) algorithm and probability generative model (PGM) for the two scenarios, respectively. Finally, our results show that the accuracy rates of KNN and PGM are beyond 90% and around 80%, respectively.**

Keywords—Hardware Security, Trojan Detection, Ring-Oscillator Network, Machine Learning, K-Nearest-Neighbor, Probability Generative Model

I. INTRODUCTION

Due to the time-to-market requirement and the increasing complexity of the Integrated Circuits (IC) design, the whole IC industry is changing to a more horizontal business mode in recent years. Thus, more system integrators are highly relying on multiple third-party intellectual properties (3PIP) and most IC companies are fabless or foundry only[1]. However, in this highly globalized supply chain, neither the 3PIP vendors nor the foundries can be trustworthy for the integrators. It is possible for 3PIP vendors to insert Trojans in design phase and the foundries may perform reverse engineering (RE) to insert Trojans in layout level. An adversary can introduce a Trojan designed to disable or destroy a system at some future time, or the Trojan could leak confidential information and secret keys covertly to the adversary[2]. This kind of threat is becoming severe in the recent decade because of the rapid globalization of IC industry. Thus, efficient Trojan detection approaches are needed.

Because most hardware Trojans are activated under rare conditions, which normally rely on some internal or input signals of the design, the traditional verification methods may not be suitable to detect the Trojans. Fuzzing test[3] has an advantage of low cost and time saving, but its input vectors are generated randomly, which will cause not enough coverage to detect Trojans. On the other aspect, exhaustive test of all signals is not practical because the chip complexity nowadays is high and a huge amount of transistors and module interconnections will be involved. Therefore, it is unlikely that these conventional test procedures make a hardware Trojan to fully activate and launch its malicious payload and observably modify the ICs' behavior[2].

The certain part of the circuits that triggers the hardware Trojans is called triggering mechanism[4] and these different mechanisms are shown in Fig. 1, which is proposed in [5].The triggering mechanism is specially defined and designed by the adversaries, thus, it can be a counter triggered after certain amount of clock cycles, or some combinational logic simply relies on a node with low switching frequency. Generally, one of the main concerns when designing them is to avoid the detection, thus, most of the conventional detection approaches are not suitable.

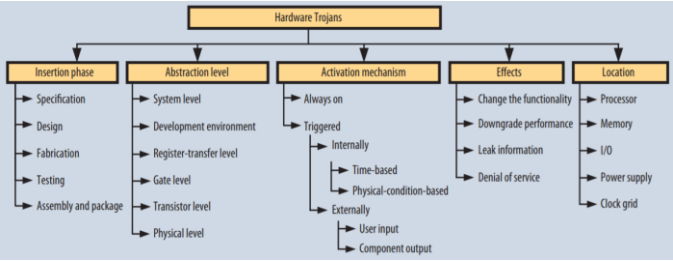

Fig. 1 Hardware Trojans taxonomy proposed in [5]

Detection methods based on side-channel analysis are practical to detection Trojans since the Trojans' activation will cause unpredicted power drop or change of clock frequency. Some approaches based on power analysis and other side-channel emissions will be discussed in Section II, in this paper, we focused on a on-chip structure called ring-oscillator network (RON)[6]. Fig. 2 shows an example architecture of the RON. In this design, the voltage switching, including the switching introduced by Trojans, in the circuits will sensitively have an impact on the frequencies of the ring-oscillators (RO). In a RON, different ring-oscillators can detect the increase in transient power consumption induced by Trojan circuits in different areas of the chip under authentication[7].

However, the real situation is more different and complicated, it is impractical to determine a settled threshold for each RO's frequency and determine if the Trojan is inserted by this threshold. In the more actual case, RO will experience a

more random variation of the frequencies. Also, process variations will influence both the Golden chips and the Trojan-inserted chips, sometimes the variations will even overshadow the frequency fluctuation that the Trojan introduced. Therefore, an overview of all the ROs' frequencies and their distribution should be considered. Also, a more robust approach with more general purpose is needed. Based on the collection of the frequency distribution in RON, we implemented two different machine learning (ML) classifiers in two scenarios to perform the Trojan detection. ML is widely focused by the researchers and industries in recent decades, leading the development of various applications[8]. Also, with the increasing demand of the mobility and performance of embedded devices, hardware implementations of the ML algorithms are also explored[9].

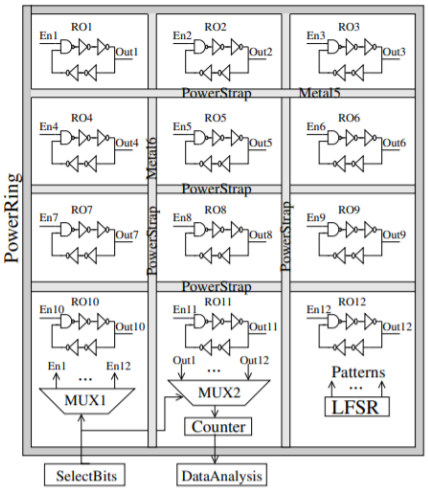

Fig. 2 An example of the RON architecture[6]

The determination of two scenarios is based on the different knowledge of the Trojan-insertion status, which will be discussed in Section III. Also, the sample configurations will be stated in Section III. In this paper, we exploited K-Nearest-Neighbor (KNN) to solve the detection task in the first scenario, and probability generative model (PGM) is utilized for the second scenario. The basic concepts and implementation details of these two algorithms will be further discussed in Section III and Section IV.

The rest of the paper is organized as follows. First, we introduced the some of the prior techniques and work in Section II. Then, in Section III, we stated the reason why we chose the KNN and PGM and a description of the concepts. In Section IV, we described the experiments and the evaluation results. Finally, we did a summary in Section V.

II. Related Work

Since the side-channel analysis is a practical direction of detecting Trojans, prior researches of detecting hardware Trojans are mainly focused on power-based analysis and critical path analysis.

Seetharam Narasimhan et al. proposed a novel non-invasive, multiple-parameter side-channel analysis based Trojan detection approach that is capable of detecting malicious hardware modifications in the presence of large process variation induced noise[10]. The internal relationship between dynamic current and maximum operating frequency is exploited and compared in different cases because the unexpected variation may be triggered by a Trojan's activation.

Dinesh Kumar Karunakaran et al. took the Gate Level Characterization (GLC) into consideration, by analyzing the leakage power, their results show that this approach detects the Trojans very accurately, especially for some tiny ones[11]. However, this approach needs the golden data to be the reference, which may be not sufficient in some cases.

In general, most of the power-based analysis may be easily influenced or even overshadowed by the process variation in the circuits. Also, these analog based approaches will cause an additional cost of the measuring and converting devices.

Some other approaches are exploited to detect the Trojans, including the critical path analyzing. Miodrag Potkonjak et al. introduced a technique for recovery of characteristics of gates in terms of leakage current, switching power, and delay, which utilizes linear programming to solve a system of equations created using nondestructive measurements of power or delays[12]. The constraint manipulation techniques can also be combined with this approach to detect hardware Trojans effectively.

Jie Li et al. discussed how a technique for precisely measuring the combinational delay of an arbitrarily large number of register-to-register paths internal to the functional portion of the IC can be used to detect the Trojans[13]. The advantage of their approach is the low cost, as well as not affecting the main function and test. However, the disadvantage is that the methods about measuring the path delay cannot handle the situation that the adversaries hid the Trojans far away from the delay-sensitive path.

Besides, there are still some attempts focusing on improving the test coverage or the test method design to activate the Trojans. The techniques introduced in [14] depends on an accidental activation of Trojans. [15] had a similar attempt, which is based on a pre-analysis of the potential locations of Trojans, then, the analytic results will help activating the Trojan more effectively. These techniques can be very difficult to accomplish in practice because they have to detect Trojans that are designed to be activated under very specific conditions[7].

III. Classifier Selection and Design

A. *Introduction to the classifiers*

This section will briefly talk about the classifier we chose and the basic introduction about how they work and their pros and cons. After talking about the algorithm, the report will illustrate the workflow, architecture of the algorithms.

The whole project will be implemented by using Python. Probabilistic generative model is a parametric approach which means we need to first assume the shape of the model. Generative model is basically based on training data and label to find the best probabilistic model which maximize the probability of a particular class. The input of probabilistic generative model is the training data with labels. For example, when doing two class classification, the model needs to generate two different probability parameters which can maximize each class's probability. Usually, we can use maximum likelihood or maximize a posterior probabilistic to find the best parameters. However, this model heavily depends on the assumption of the

model shape. The testing phase is to input the test data and compare the posterior probabilities of these two models.

Traditional parametric approaches have drawbacks that parametric form needs to be assumed or decided in advance and, if chosen poorly, might be a poor model of the distribution that generates the data resulting in poor performance. Non-parametric approaches are those that do not assume a particular generating distribution for the data.

KNN is a supervised clustering algorithm which always used in regression and classification problems.

Nearest neighbor methods compare a test point to the k nearest training data points and then estimate an output value based on the desired output values of the k nearest training points. Therefore, basically, KNN does not have the training phase other than storing the training data points and the label.

When doing the test phase, you need to first determine which k training data points are closest to the test point and meanwhile determine the output value for the test point. In usual, KNN always use Euclidian distance calculated by (1).

$$d_E = \sqrt{(x_1 - x_2)^T (x_1 - x_2)} \quad (1)$$

B. *Workflow and architecture of the algorithms*

Fig. 3 shows the workflow of KNN algorithm, KNN uses the distance of different training data points of different classes and testing data to call a voting process. The winner will be the class of the test point. The decision is made by considering the number of neighbors that each test data point besides. Therefore, the hyper parameter k basically means how many nearest neighbors the model based on to do classification is crucial.

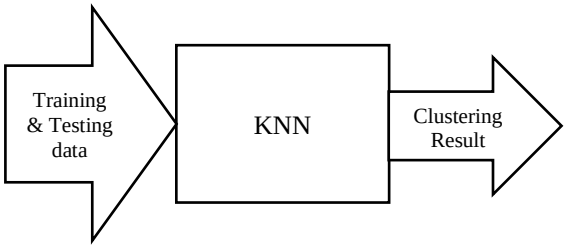

Fig. 3 Block diagram of the KNN

The main idea of the PGM is to compare the probability or, in another word, build a discriminative model of the classes. Fig. 4 shows the block diagram of PGM. The workflow of PGM is first to use training data set to train the probability model. For example, if we assume the model's shape is gaussian function, the parameters of gaussian like covariance and expectation will be trained in the training process. The testing process is to bring test data into the model and compare the posterior probability which means which model has a bigger probability, the current test data are more prone to that class.

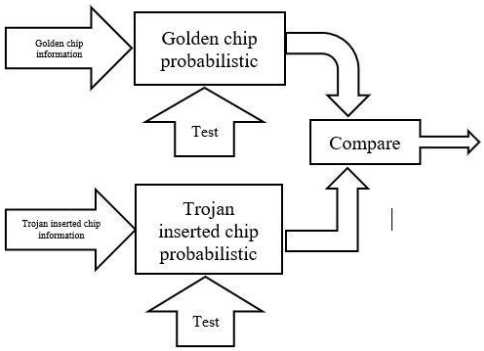

Fig. 4 Block diagram of the PGM

IV. CLASSIFIER EVALUATION RESULTS

A. *Implementation*

This part is separated by two scenarios. First scenario is the implementation when we have all the information of Golden chips and Trojan inserted chips. Second scenario is when only have the information of golden chips.

In these two scenarios, the implementation of supervised and unsupervised learning methods will be implemented. This part basically covers all the implementation including data preprocessing, training phase and testing phase.

In scenario A, both the information of the Golden chips and Trojan inserted chips are given. Therefore, we can use supervised learning to do classification. In this scenario, the KNN algorithm will be implemented. Because there is no training process in KNN, so the training and testing process of KNN is included in a single model. The output after KNN is the clustering result which can be used as evaluate the performance of the model. Cross validation is implemented during the data preprocessing phase. The purpose of cross validation is to reduce overfitting. The cross validation will be implemented by using 6 samples, 12 samples and 24 samples to as training data and the rest to use as testing data.

In scenario B, only the information of both Golden chips is provided. In this case, we can also use supervised learning to train the model. We use probabilistic generative model to do classification. When using probabilistic generative model, because this is a two classes classification problem, two different probability density functions will be trained by using the training data. All the data will be firstly separated into two parts in order to do cross validation in the data preprocessing part. Both training data of the golden chips and the inserted chips will be fed into the model and generate the discriminative model after finished the training part. During the testing part, the test data will be fed into the model and go through the compare part online. The cross validations will be implemented by using 6 samples, 12 samples and 24 samples differently.

B. *Results analysis*

Fig. 5 shows the accuracy rate comparison of KNN(K=4) algorithm when using 6, 12 and 24 samples to do training. We can see from this figure that the accuracy of most Trojans detection is higher than 90%. TABLE I shows the confusion matrix of KNN. The true positive rate of KNN is 95% and the false positive rate is 41%.

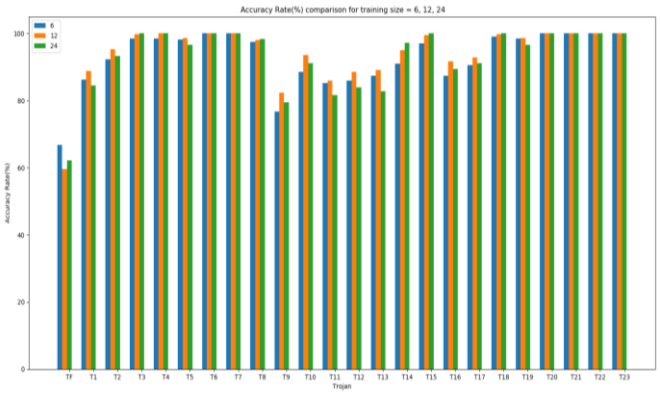

Fig. 5 Accuracy rate comparison of KNN algorithm

TABLE 1. CONFUSION MATRIX OF KNN

	Condition Positive	Condition Negative
Predict positive	11811	444
Predict negative	609	636

In scenario B, the TPR and FPR of probability generative model are shown by Fig. 6 and Fig. 7. It shows that the TPR of probability generative model is about 90% and FPR of the generative model is below 50%. Meanwhile, Fig. 8 shows the accuracy of the generative model. The accuracy of the model is about 80% which is reasonable because in scenario B, we only have the information about the golden chip.

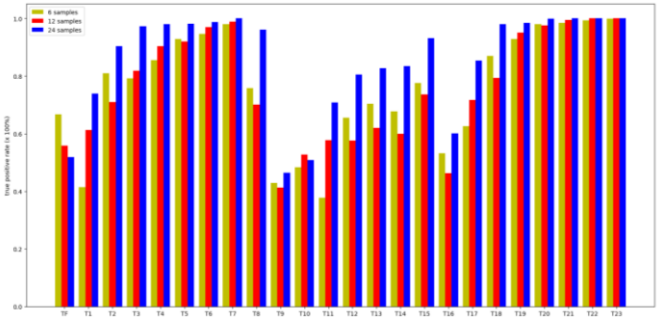

Fig. 6 True Positive Rate of PGM

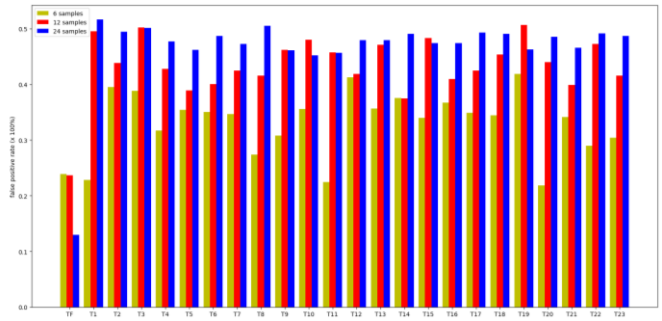

Fig. 7 False Positive Rate of PGM

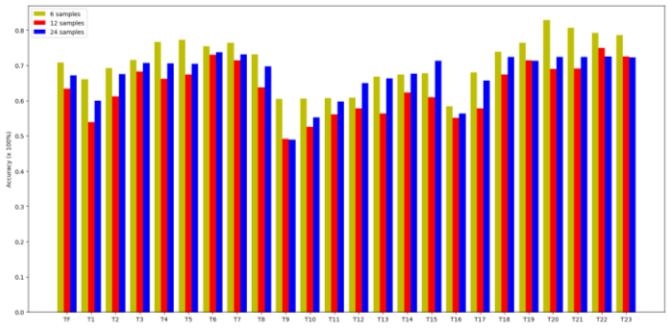

Fig. 8 Accuracy of PGM

V. CONCLUSION

This paper performed two ML-based classifiers, including KNN and PGM, to support the hardware Trojan detection on RON. First, we discussed the motivations, current problems and the prior researches of Trojan detection. Then, we introduced the details about these two algorithms and the reason why we chose them. Next, we described the work flow of our experiment and evaluation. Our results show that for KNN in scenario A, the accuracy rate is higher than 90%, the TPR is 95% and the FPR is 41%. For PGM in scenario B, the accuracy rate is around 80%, the TPR is 90% and the FPR is below 50%.

In general, we can draw a conclusion that our work is a high accuracy implementation of the ML-based hardware Trojan detection. Also, it proves that these two algorithms are suitable for the classification task of RON frequencies. Besides, we can observe that the accuracy of scenario B is lower because only the Golden data are used for training. In the future work, one direction is that we can further optimize our current implementations to have a more accurate result, another is that more ML algorithms can be explored to support Trojan detection.

REFERENCES

[1] S. Bhunia, M. Abramovici, D. Agrawal, P. Bradley, M. Hsiao, J. Plusquellic, and M. Tehranipoor, “Protection against hardware trojan attacks: Towards a comprehensive solution,” *Design Test, IEEE*, vol. 30, no. 3, pp. 6–17, June 2013.

[2] M. Tehranipoor and F. Koushanfar, "A Survey of Hardware Trojan Taxonomy and Detection," in IEEE Design & Test of Computers, vol. 27, no. 1, pp. 10-25, Jan.-Feb. 2010.

[3] S. Duncan, "Fuzzing For Software Security Testing and Quality Assurance," Software Quality Professional, vol. 11, (3), pp. 47-48, 2009.

[4] R. S. Chakraborty, S. Paul and S. Bhunia, "On-demand transparency for improving hardware Trojan detectability," 2008 IEEE International Workshop on Hardware-Oriented Security and Trust, Anaheim, CA, 2008, pp. 48-50.

[5] R. Karri, J. Rajendran, K. Rosenfeld and M. Tehranipoor, "Trustworthy Hardware: Identifying and Classifying Hardware Trojans," in Computer, vol. 43, no. 10, pp. 39-46, Oct. 2010.

[6] X. Zhang and M. Tehranipoor, "RON: An on-chip ring oscillator network for hardware Trojan detection," 2011 Design, Automation & Test in Europe, Grenoble, 2011, pp. 1-6.

[7] Shane Kelly, Xuehui Zhang, Mohammed Tehranipoor, and Andrew Ferraiuolo, "Detecting Hardware Trojans using On-chip Sensors in an ASIC Design." Journal of Electronic Testing 31, no. 1 (2015): 11-26.

[8] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” Nature, vol. 521, no. 7553, pp. 436–444, May 2015.

[9] S. Cadambi et al., "A Massively Parallel FPGA-Based Coprocessor for Support Vector Machines," 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines, Napa, CA, 2009, pp. 115-122.

[10] S. Narasimhan et al., "Multiple-parameter side-channel analysis: A non-invasive hardware Trojan detection approach," 2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), Anaheim, CA, 2010, pp. 13-18.

[11] D. K. Karunakaran and N. Mohankumar, "Malicious combinational Hardware Trojan detection by Gate Level Characterization in 90nm technology," Fifth International Conference on Computing, Communications and Networking Technologies (ICCCNT), Hefei, 2014, pp. 1-7.

[12] M. Potkonjak, A. Nahapetian, M. Nelson and T. Massey, "Hardware Trojan horse detection using gate-level characterization," 2009 46th ACM/IEEE Design Automation Conference, San Francisco, CA, 2009, pp. 688-693.

[13] Jie Li and J. Lach, "At-speed delay characterization for IC authentication and Trojan Horse detection," 2008 IEEE International Workshop on Hardware-Oriented Security and Trust, Anaheim, CA, 2008, pp. 8-14.

[14] S. Jha and S. K. Jha, "Randomization Based Probabilistic Approach to Detect Trojan Circuits," 2008 11th IEEE High Assurance Systems Engineering Symposium, Nanjing, 2008, pp. 117-124.

[15] M. Banga and M. S. Hsiao, "A region based approach for the identification of hardware Trojans," 2008 IEEE International Workshop on Hardware-Oriented Security and Trust, Anaheim, CA, 2008, pp. 40-47.

CODE:

project_KNN.py

```
""" ======================  Import dependencies
========================= """
import numpy as np
from matplotlib import pyplot as plt
from sklearn.model_selection import
train_test_split
from sklearn import neighbors
import pandas as pd
import random
import csv
from time import time

""" ======================  Function definitions
========================== """
def SplitData(data, num):
    number = np.arange(33)
    train_num = random.sample(number.tolist(),
num)
    data_true_train = np.zeros((num*2, 8))
    data_false_train = np.zeros((num*23, 8))
    data_true_test = np.zeros(((33 - num)*2, 8))
    data_false_test = np.zeros(((33 - num)*23, 8))
    n1 = 0
    m1 = 0
    n2 = 0
    m2 = 0
    for i in range(33):
        if i in train_num:
            data_true_train[n1:n1 + 2, :] =
data[i*2:i*2 + 2, :]
            data_false_train[m1:m1 + 23, :] =
data[66 + i*23:66 + i*23 + 23, :]
            n1 = n1 + 2
            m1 = m1 + 23
        else:
            data_true_test[n2:n2 + 2, :] =
data[i*2:i*2 + 2, :]
            data_false_test[m2:m2 + 23, :] =
data[66 + i*23:66 + i*23 + 23, :]
            n2 = n2 + 2
            m2 = m2 + 23

    data_train = np.vstack((data_true_train,
data_false_train))
    data_test = np.vstack((data_true_test,
data_false_test))
    target_train = np.hstack((np.zeros(num*2),
np.ones(num*23)))
    target_test = np.hstack((np.zeros(66 - num*2),
np.ones(759 - num*23)))
    return data_train, target_train, data_test,
target_test

def ACC(pre, target_test):
    tp_KNN = 0
    fp_KNN = 0
    fn_KNN = 0
    tn_KNN = 0
    for i in range(len(target_test)):
        if pre[i] == 1:
            if pre[i] == target_test[i]:
                tp_KNN = tp_KNN + 1
            else:
                fp_KNN = fp_KNN + 1
        else:
            if pre[i] == target_test[i]:
                tn_KNN = tn_KNN + 1
            else:
                fn_KNN = fn_KNN + 1
    acc = (tp_KNN + tn_KNN)/(len(target_test))
    return acc

def calmeanacc(trainnum):
    trainnum = trainnum
    testnum = 33 -trainnum
    n_neighbors = 4
    acc = np.zeros((20, 24))
    meanacc = np.zeros(24)
    for i in range(20):
        data_train, target_train, data_test,
target_test = SplitData(data, trainnum)
        classifiers =
neighbors.KNeighborsClassifier(n_neighbors,
weights='distance')
        clffinal = classifiers.fit(data_train,
target_train)
        pre = clffinal.predict(data_test)
        for j in range(testnum*2):
            if pre[j] == target_test[j]:
                acc[i,0] = acc[i,0] + 1
        for n in range(testnum):
            for m in range(23):
                if pre[testnum*2 + m + n*23] ==
target_test[testnum*2 + m + n*23]:
                    acc[i, m + 1] = acc[i, m + 1]
+ 1

    acc[:,0] = acc[:,0]/(testnum*2)
    acc[:,1::] = acc[:,1::]/testnum
    for i in range(24):
        meanacc[i] = acc[:, i].mean()
    return meanacc

def calruntime(trainnum):
    n_neighbors = 4
    tp_KNN = 0
    fp_KNN = 0
    fn_KNN = 0
    tn_KNN = 0
    time_train = np.zeros(20)
    time_test = np.zeros(20)
    for i in range(20):
        data_train, target_train, data_test,
target_test = SplitData(data, trainnum)

        train_start = time()
        classifiers =
neighbors.KNeighborsClassifier(n_neighbors,
weights='distance')
        clffinal = classifiers.fit(data_train,
target_train)
        train_end = time()
        time_train[i] = train_end - train_start

        test_start = time()
        pre = clffinal.predict(data_test)
```

```
        test_end = time()
        time_test[i] = test_end - test_start

    mean_train_time = time_train.mean()
    mean_test_time = time_test.mean()
    return mean_train_time, mean_test_time

def calall(trainnum):
    trainnum = trainnum
    testnum = 33 -trainnum
    n_neighbors = 4
    acc = np.zeros((20, 24))
    allacc = np.zeros((4,24))
    for i in range(20):
        data_train, target_train, data_test,
target_test = SplitData(data, trainnum)
        classifiers =
neighbors.KNeighborsClassifier(n_neighbors,
weights='distance')
        clffinal = classifiers.fit(data_train,
target_train)
        pre = clffinal.predict(data_test)
        for j in range(testnum*2):
            if pre[j] == target_test[j]:
                acc[i,0] = acc[i,0] + 1
        for n in range(testnum):
            for m in range(23):
                if pre[testnum*2 + m + n*23] ==
target_test[testnum*2 + m + n*23]:
                    acc[i, m + 1] = acc[i, m + 1]
+ 1

    acc[:,0] = acc[:,0]/(testnum*2)
    acc[:,1::] = acc[:,1::]/testnum
    for i in range(24):
        allacc[0, i] = acc[:, i].mean()*100
        allacc[1, i] = acc[:, i].std()*100
        allacc[2, i] = acc[:, i].max()*100
        allacc[3, i] = acc[:, i].min()*100
    return allacc

""" ======================  Load Training Data
====================== """
data = np.zeros((33, 25, 8))
data_true = []
data_false = []
for i in range (33):
    filename = 'ROFreq\\Chip' + str(i + 1) +
'.xlsx'
    datatemp = pd.read_excel(filename, header =
None)
    datatemp = np.array(datatemp)
    data[i,:,:] = datatemp
    data_true.append(datatemp[0,:])
    data_true.append(datatemp[24,:])
    for j in range(23):
        data_false.append(datatemp[j + 1,:])

data =
np.vstack((np.array(data_true),np.array(data_false
)))
target = np.hstack((np.zeros(66), np.ones(759)))

""" ======================  Train and Test KNN
Model and Calculate TP/FP/TN/FN
====================== """
trainnum = 6
n_neighbors = 4
tp_KNN = 0
fp_KNN = 0
fn_KNN = 0
tn_KNN = 0
time_train = np.zeros(20)
time_test = np.zeros(20)
for i in range(20):
    data_train, target_train, data_test,
target_test = SplitData(data, trainnum)
    classifiers =
neighbors.KNeighborsClassifier(n_neighbors,
weights='distance')
    clffinal = classifiers.fit(data_train,
target_train)
    pre = clffinal.predict(data_test)
    for i in range(len(target_test)):
        if pre[i] == 1:
            if pre[i] == target_test[i]:
                tp_KNN = tp_KNN + 1
            else:
                fp_KNN = fp_KNN + 1
        else:
            if pre[i] == target_test[i]:
                tn_KNN = tn_KNN + 1
            else:
                fn_KNN = fn_KNN + 1

""" ======================  Calculate Run Time
and Plot ====================== """
train_time = np.zeros(3)
test_time = np.zeros(3)

train_time[0], test_time[0] = calruntime(6)
train_time[1], test_time[1] = calruntime(12)
train_time[2], test_time[2] = calruntime(24)

plt.figure(1)
x_run = ['6', '12', '24']
plt.bar(np.arange(3) - 0.1, train_time, width =
0.2, tick_label = x_run, label = 'Training Time')
plt.bar(np.arange(3) + 0.1, test_time, width =
0.2, tick_label = x_run, label = 'Evaluation
Time')
plt.xlabel('Training Size')
plt.ylabel('Run Time(s)')
plt.title('Average Training Time and Evaluation
Time')
plt.legend()

""" ========================  Figure Accuracy
========================== """
all6 = calall(6)
all12 = calall(12)
all24 = calall(24)
np.savetxt("6.csv", all6, delimiter=",")
np.savetxt("12.csv", all12, delimiter=",")
np.savetxt("24.csv", all24, delimiter=",")

""" ========================  Plot Accuracy
========================== """
accforplot = np.zeros((3, 24))
```

```
accforplot[0,:] = calmeanacc(6) * 100
accforplot[1,:] = calmeanacc(12) * 100
accforplot[2,:] = calmeanacc(24) * 100
name_list = ['TF', 'T1', 'T2', 'T3', 'T4', 'T5',
'T6', 'T7', 'T8', 'T9', 'T10', 'T11', 'T12',
'T13', 'T14', 'T15', 'T16', 'T17', 'T18', 'T19',
'T20', 'T21', 'T22', 'T23']

plt.figure(2)
plt.bar(np.arange(24), accforplot[0,:], width =
0.2, tick_label = name_list, label = '6')
plt.bar(np.arange(24) + 0.2, accforplot[1,:],
width = 0.2, tick_label = name_list, label = '12')
plt.bar(np.arange(24) + 0.4, accforplot[2,:],
width = 0.2, tick_label = name_list, label = '24')
plt.xlabel('Trojan')
plt.ylabel('Accuracy Rate(%)')
plt.title('Accuracy Rate(%) comparison for
training size = 6, 12, 24')
plt.legend()
plt.show()
```

Project_PGM.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
import xlrd
import random
import datetime

def prob(train,test, threshold):
    # calculate the mean and cov
    #input: row vector
    #train
    start = datetime.datetime.now()
    mean = np.mean(train, axis=0)
    cov = np.cov(train.T)
    # label the test data
    prob_class1 = multivariate_normal.pdf(test,
mean=mean, cov=cov,allow_singular=True)
    end = datetime.datetime.now()
    print("Train time used:", end - start)
    start = datetime.datetime.now()
    label_pos = []
    for k_label in range(0, test.shape[0]):
        if prob_class1[k_label] < threshold:
            labelk = 1
        else:
            labelk = 0
        label_pos.append(labelk)
    end = datetime.datetime.now()
    print("Test time used:", end - start)
    return label_pos



def loaddata():
    TF = np.zeros((1,8))
    TI = np.zeros((1,8))
    for i in range(1,34):
        name = 'Chip%d.xlsx'%i
        data = xlrd.open_workbook(name)
        table = data.sheet_by_index(0)
        TF = np.vstack((TF,
np.array(table.row_values(0))))
        TF = np.vstack((TF,
np.array(table.row_values(24))))
        for j in range(1,24):
            TI = np.vstack((TI,
np.array(table.row_values(j))))
    return TF[1:, :], TI[1:, :]

def
test(TF,TI,sample_number,threshold,row_number):
    index = range(0,66)
    index = np.array(index)
    random.shuffle(index)
    train = np.zeros((sample_number,8))
    test_TI = np.zeros((33,8))
    test_TF = np.zeros((66-sample_number,8))
    for i in range(0,sample_number):
        train[i,:] = TF[index[i],:]
    for i in range(0, 66-sample_number):
        test_TF[i,:] =
TF[index[i+sample_number],:]
    for i in range(0,33):
        test_TI[i,:] = TI[row_number+i*23,:]
    lable_predict_TI = prob(train, test_TI,
threshold)
    lable_predict_TF = prob(train, test_TF,
threshold)
    TPR =
sum(lable_predict_TI)/len(lable_predict_TI)
    FPR = sum(lable_predict_TF) /
len(lable_predict_TF)
    accuracy =
(sum(lable_predict_TI)+len(lable_predict_TF)-
sum(lable_predict_TF))/(len(lable_predict_TI)+len(
lable_predict_TF))
    return TPR,FPR,accuracy



'''calculate the TPR, FPR and accuracy'''
TF,TI = loaddata()
TPR_track = np.zeros((3,23))
FPR_track = np.zeros((3,23))
TPR_TF = np.zeros((3,23))
FPR_TF = np.zeros((3,23))
Accuracy = np.zeros((3,23))
for row in range(0,23):
    # 6 samples
    TPR = []
    FPR = []
    Accu = []
    for i in range (0,25):
        tpr,fpr,accuracy = test(TF,TI,6,10**(-
7),row)
        TPR.append(tpr)
        FPR.append(fpr)
        Accu.append(accuracy)
    ave_tpr = np.mean(TPR)
    ave_fpr = np.mean(FPR)
    std_tpr = np.std(TPR)
    std_fpr = np.std(FPR)
    max_tpr = np.max(TPR)
    max_fpr = np.max(FPR)
```

```
    Accuracy[0,row] = np.mean(Accu)
    TPR_track[0, row] = ave_tpr
    FPR_track[0, row] = ave_fpr
    TPR_TF[0, row] = 1-ave_fpr
    FPR_TF[0, row] = 1 - ave_tpr



    # 12 samples
    TPR = []
    FPR = []
    Accu = []
    for i in range (0,25):
        tpr,fpr,accuracy = test(TF,TI,12,10**(-
10),row)
        TPR.append(tpr)
        FPR.append(fpr)
        Accu.append(accuracy)
    ave_tpr = np.mean(TPR)
    ave_fpr = np.mean(FPR)
    std_tpr = np.std(TPR)
    std_fpr = np.std(FPR)
    max_tpr = np.max(TPR)
    max_fpr = np.max(FPR)
    Accuracy[1, row] = np.mean(Accu)
    TPR_track[1, row] = ave_tpr
    FPR_track[1, row] = ave_fpr
    TPR_TF[1, row] = 1 - ave_fpr
    FPR_TF[1, row] = 1 - ave_tpr

    # 24 samples
    TPR = []
    FPR = []
    Accu = []
    for i in range (0,25):
        tpr,fpr,accuracy = test(TF,TI,24,10**(-
6),row)
        TPR.append(tpr)
        FPR.append(fpr)
        Accu.append(accuracy)
    ave_tpr = np.mean(TPR)
    ave_fpr = np.mean(FPR)
    std_tpr = np.std(TPR)
    std_fpr = np.std(FPR)
    max_tpr = np.max(TPR)
    max_fpr = np.max(FPR)
    Accuracy[2, row] = np.mean(Accu)
    TPR_track[2, row] = ave_tpr
    FPR_track[2, row] = ave_fpr
    TPR_TF[2, row] = 1 - ave_fpr
    FPR_TF[2, row] = 1 - ave_tpr

Accuracy_tf_0 = np.mean(Accuracy[0,:])
Accuracy_tf_1 = np.mean(Accuracy[1,:])
Accuracy_tf_2 = np.mean(Accuracy[2,:])
tpr_tf_0 = np.mean(TPR_TF[0,:])
fpr_tf_0 = np.mean(FPR_TF[0,:])
tpr_tf_1 = np.mean(TPR_TF[1,:])
fpr_tf_1 = np.mean(FPR_TF[1,:])
tpr_tf_2 = np.mean(TPR_TF[2,:])
fpr_tf_2 = np.mean(FPR_TF[2,:])
'''plot the TPR, FPR, and Accuracy figures'''
# TPR figure
plt.figure(1)
name_list =
['TF','T1','T2','T3','T4','T5','T6','T7','T8','T9'
,'T10','T11','T12','T13','T14','T15','T16','T17','
T18','T19','T20','T21','T22','T23']
num_list1 = np.append(tpr_tf_0, TPR_track[0,:])
num_list2 = np.append(tpr_tf_1, TPR_track[1,:])
num_list3 = np.append(tpr_tf_2, TPR_track[2,:])
x =list(range(len(num_list1)))
total_width, n = 0.8, 3
width = total_width / n
plt.bar(x, num_list1, width=width, label='6
samples',fc = 'y')
for i in range(len(x)):
    x[i] = x[i] + width
plt.bar(x, num_list2, width=width, label='12
samples',tick_label = name_list,fc = 'r')
for i in range(len(x)):
    x[i] = x[i] + width
plt.bar(x, num_list3, width=width, label='24
samples',fc = 'b')
plt.ylabel('true positive rate (x 100%)')
plt.legend()
# FPR figure
plt.figure(2)
name_list =
['TF','T1','T2','T3','T4','T5','T6','T7','T8','T9'
,'T10','T11','T12','T13','T14','T15','T16','T17','
T18','T19','T20','T21','T22','T23']
num_list1 = np.append(fpr_tf_0, FPR_track[0,:])
num_list2 = np.append(fpr_tf_1, FPR_track[1,:])
num_list3 = np.append(fpr_tf_2, FPR_track[2,:])
x =list(range(len(num_list1)))
total_width, n = 0.8, 3
width = total_width / n
plt.bar(x, num_list1, width=width, label='6
samples',fc = 'y')
for i in range(len(x)):
    x[i] = x[i] + width
plt.bar(x, num_list2, width=width, label='12
samples',tick_label = name_list,fc = 'r')
for i in range(len(x)):
    x[i] = x[i] + width
plt.bar(x, num_list3, width=width, label='24
samples',fc = 'b')
plt.ylabel('false positive rate (x 100%)')
plt.legend()
# Accuracy figure
plt.figure(3)
name_list =
['TF','T1','T2','T3','T4','T5','T6','T7','T8','T9'
,'T10','T11','T12','T13','T14','T15','T16','T17','
T18','T19','T20','T21','T22','T23']
num_list1 = np.append(Accuracy_tf_0,
Accuracy[0,:])
num_list2 = np.append(Accuracy_tf_1,
Accuracy[1,:])
num_list3 = np.append(Accuracy_tf_2,
Accuracy[2,:])
x =list(range(len(num_list1)))
total_width, n = 0.8, 3
width = total_width / n
plt.bar(x, num_list1, width=width, label='6
samples',fc = 'y')
for i in range(len(x)):
    x[i] = x[i] + width
plt.bar(x, num_list2, width=width, label='12
```

```
samples',tick_label = name_list,fc = 'r')
for i in range(len(x)):
    x[i] = x[i] + width
plt.bar(x, num_list3, width=width, label='24
samples',fc = 'b')
plt.ylabel('Accuracy (x 100%)')
plt.legend()
plt.show()
```