Discriminer: A tool of discriminating program traces (runs) based on observations

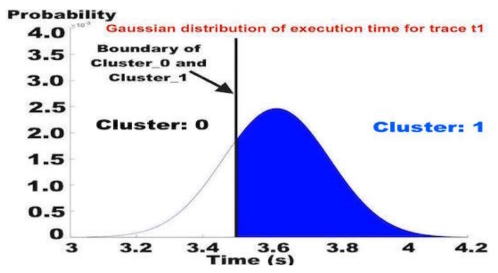
Goal: Program Confidentiality Analysis in presence of Side-Channel

Requirement:

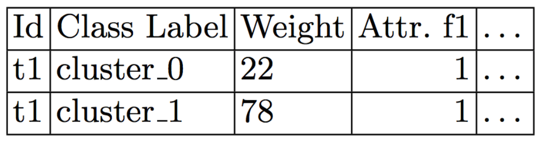
Python 2.7 or more
Libraries: Numpy, Pandas, matplotlib, sklearn, and scipy

Overall Description

Dicriminer is a combination of program analysis and machine learning tool which helps users to learn a classification model about program traces. First, a program is analyzed and input data is obtained using dynamic analysis. The data features can be function calls, basic block, branches and so on. In addition, the data should include an observable feature like time of execution or memory consumption for each record (each record of data is equal to some features of a program trace). Note that due to probabilistic modeling, the data should include 10 times of observations for the same record, or it may include mean and standard deviation for each record. Given this data as input, the clustering part will divide the data into numbers of clusters (specified by the user) based on distance measures like euclidean distance using the mean of each record. As a result, each record is assigned to a cluster based on its mean. In the next step, the clustering procedure considers a stochastic distribution for each observation, and it calculates a probability (weight) to be assigned to different clusters for all records. For example, suppose t1 is a trace which has time observation such that the mean of execution time is 3.6 seconds and standard deviation is 161 as follow:



(a)



Id	Class Label	Weight	Attr. f1	...
t1	cluster_0	22	1	...
t1	cluster_1	78	1	...

(b)

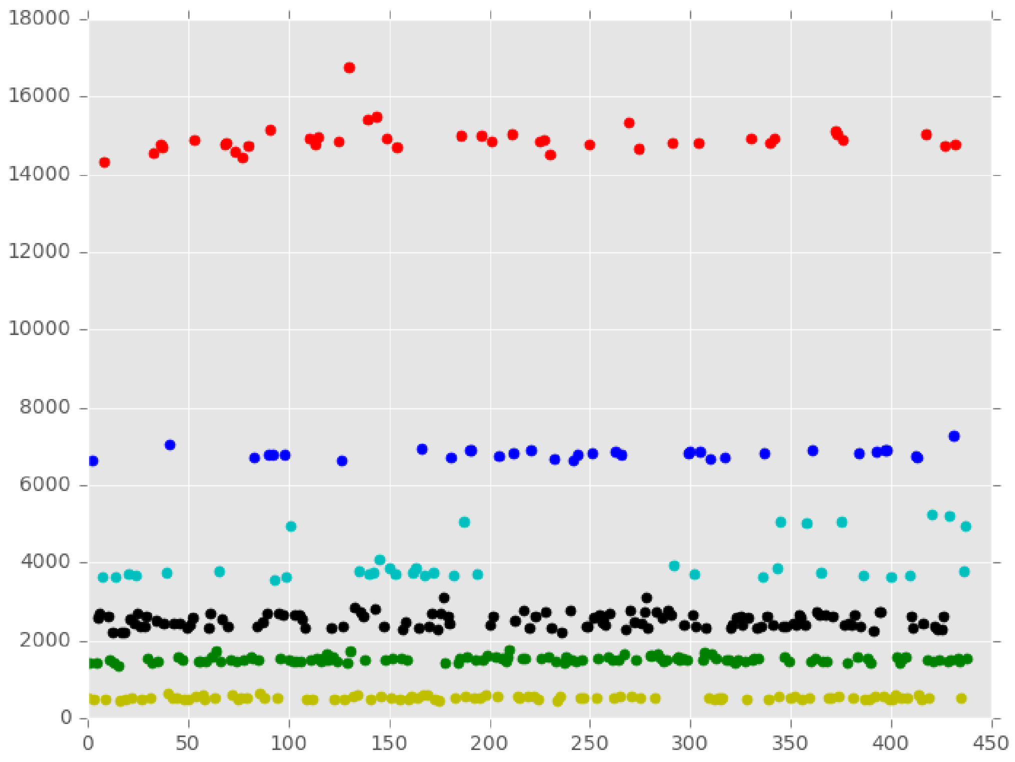
The clustering based on mean will assign it to Cluster_1, but due to probabilistic behaviors of time (assume that time is normal random variable), we also consider the probability to be assigned to other clusters like Cluster_0 and add weight feature to each record to show the probability. The output data of clustering step will include inside features of the program like function calls, label, and weight for each record. This data will be input for classification step in which a classifier model like decision tree will learn a model which classifies (discriminates) traces. As each label is equal to a cluster (for example a time range of execution), the classifier distinguishes traces based on the observations. If the inside features are function calls, for instance, the classifier will learn models like if function f1() is called, then the trace should be assigned to label 1. As label 1 shows a range of time like [t1,t2], the model predicts that calling function f1 in a trace will determine the execution of program trace between [t1,t2]. This observation can lead to confidentiality violation as described in the following: Discriminating trace using Time!

Data Extraction

The input data for Discriminer should put inside **Clustering_input**. The format of the file should be **.csv**. In order to extract data from a target program, you can use standard dynamic tools like **Soot**. However, these tools are very broad and support many functions. We will provide you a compact tool which collects necessary data from a target program (coming soon). The input data in **.csv** format should have three main parts: 1- ID for each record (unique number) 2- 10 features named **"T1"**...**"T10"** which set to values of 10 measurements of the same record. For example, you may execute a program with the same input 10 times, and **T1**...**T10** are time recorded in each execution. This is one way to introduce observations like time to the input data set. As another way, you can include two features named **"mean"** and **"std"** which are the mean and standard deviation for each record. If your records are constant, you can set all **"T1"**...**"T10"** to constant value, or if you use **"mean"** and **"std"** features, the mean feature should set to the constant value and standard deviation should set to 0. 3- Features about program inside behaviors. For example, these features can be function names, and their values can be the number of times the functions are called. You can find some examples of the input data inside **Clustering_input**. These data include both ways to represent the behaviors (using either **"T1"** ...**"T10"** or **"mean"** **"std"**).

Data Clustering (label and weight calculations)

After putting data inside **Clustering_input**, you can run **Cluster.py** to cluster data and calculate required features for classification step (to run: python Cluster.py). The program asks users about the name of the input file (just enter the file name without any relative or absolute address and any type). Then, it will ask whether you have the features **"T1"**...**"T10"**. If the answer is "no", you should put **"mean"** and **std** features inside the data. Then, it will ask about the name of output files namely clustering plot name and clustering result data set. The following is the result of clustering plot for a benchmark named SnapBuddy_2 (axis y is mean of execution time and axis x is data ID):



As explained earlier, **"mean"** feature is the distance measurement to cluster data. So. the plot shows data clustering based on the mean measurement of each record. Then, the program calculates boundaries between clusters. The boundaries are calculated as follow: first, it sorts all data inside different clusters. Then, it picks the maximum value of cluster n and the minimum value of cluster n+1. The boundary line is the average of these two numbers. In the case of t1, the boundary line between cluster_0 and cluster_1 is about 3.5 seconds. Considering the measurement (like time) is normal random variable, we calculate the probability of a record to assign all clusters. For this aim, we calculate **Cumulative distribution function** of normal distribution in every cluster boundaries. For example, cfd of t1 is 0.78 to be assigned to Cluster_1 and 0.22 in the case of Cluster_0. So, t1 will replicate two times with the same ID. One labeled Clsuter_0 and weighted 22 and another labeled Cluster_1 and weighted 78. The rest of the features will be the same for both of them. In this fashion, the output of clustering procedure will be calculated and fed to classification step (as input). The output file will go into **Clustering_results** in **.csv** format. For classification step, you need to copy the result file inside **Clustering_results** into **Classification_input**.

Learning Classifier

The classification procedure considers input file to be inside **Classification_input**. You can run the program by typing following command: python Classify.py The program asks about input file name (name which is inside **Classification_input** folder without any type). As it uses K-fold cross_validation to split learning and testing data, the program asks you to enter K. Note that if you press enter without any number, the default value will be considered. As the classifier is a decision tree, the program asks if the user prefers to bound the depth of decision tree to a constant value. The program applies decision tree learning model and calculates accuracy for the model. Finally, it will produce three top accurate decision trees in **.dot** format inside **Classification_results** and print out the accuracy of each tree at the end of program execution. Users can enter the following command in terminal to produce final trees (for example, suppose the output file is cluster_result_data_gabfeed2_tree0.dot):
dot -Tpng Classification_results/cluster_result_data_gabfeed2_tree0.dot -o Classification_results/cluster_result_data_gabfeed2_tree0.png
And the result decision tree will look like the following (partial decision tree, see the full one inside **Classification_results**):

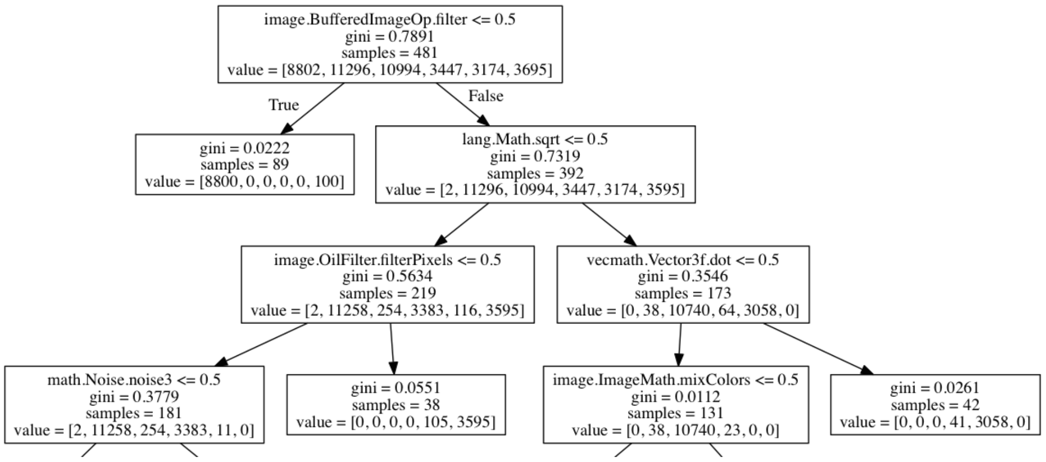


image.BufferedImageOp.filter <= 0.5
gini = 0.7891
samples = 481
value = [8802, 11296, 10994, 3447, 3174, 3695]
True
False
gini = 0.0222
samples = 89
value = [8800, 0, 0, 0, 0, 100]
lang.Math.sqrt <= 0.5
gini = 0.7319
samples = 392
value = [2, 11296, 10994, 3447, 3174, 3595]
image.OilFilter.filterPixels <= 0.5
gini = 0.5634
samples = 219
value = [2, 11258, 254, 3383, 116, 3595]
vecmath.Vector3f.dot <= 0.5
gini = 0.3546
samples = 173
value = [0, 38, 10740, 64, 3058, 0]
math.Noise.noise3 <= 0.5
gini = 0.3779
samples = 181
value = [2, 11258, 254, 3383, 11, 0]
gini = 0.0551
samples = 38
value = [0, 0, 0, 0, 105, 3595]
image.ImageMath.mixColors <= 0.5
gini = 0.0112
samples = 131
value = [0, 38, 10740, 23, 0, 0]
gini = 0.0261
samples = 42
value = [0, 0, 0, 41, 3058, 0]

The decision tree can be seen as a set of discrimination formulae. For example, it says that if **"image.BufferedImageOp.filter"** is not called, then the record belongs to Cluster_0 (As it has much more samples in comparison to other clusters). The another example is the one which assigns records to last cluster namely Cluster_5. If **"image.BufferedImageOp.filter"** and **"image.OilFilterPixel"** are called, while **"lang.Math.sqrt"** is not called, then the data record should be assigned to Cluster_5.