Adversarial Machine Learning against keystroke dynamics

Negi, Parimarjan
Stanford University
pnegi@stanford.edu

Sharma, Ankita
Stanford University
ankita89@stanford.edu

***Abstract*—The paper starts with an introduction and purpose of this project (I), relevant past research (II), and then goes on to explain the data set used and explain features and preprocessing (III). We then explain methodology and results of the anomaly detection part (IV.A) followed by part of attacking the classifiers (IV.B). The observations are summarized in section V and briefly explain the possible future work related to the project.**

I. INTRODUCTION

Keystroke dynamics is the study of detailed timing information about a persons typing patterns. This information can be used to construct additional security measures for various online sites without adding any inconvenience to the users. It has actually already been implemented in practice in various banks - for instance, many Norwegian banks have been using this since 2014 [8].

The security layer using such a system is targeted for cases where the attacker has access to a users password but has no other information about the user - so he cant use approaches such as installing malware on the users computers to generate their typing profiles. This case can happen quite commonly - for example, if the passwords from a large website are compromised (like the massive LinkedIn password breach from 2012).

The classifier takes in the labelled keystroke timing patterns for 51 users and classifies (detailed in section 4) them into valid or fraudulent attempts. Attackers generate adversarial keystroke samples that make an otherwise robust classifier accept the artificially generated samples as belonging to the valid user.

II. LITERATURE REVIEW

Most of the research literature has focused on building robust classifiers. For training our classifiers, we based our models on the recent work by Monaco [1] for Keystroke Biometrics Ongoing Competition (KBOC) [2] But the goals of KBOC are clearly different from ours - which led to us adopting a very different framework to set up our code base and dataset which we compare in section III. In the adversarial setting, a research paper with similar goals was Serwadda and co. [5]. There they started from one of the impostor samples and used Gaussian perturbations to eventually try crack the classifiers - but these worked well only for a small subset of users.

III. DATA SET AND FEATURES

Following are the reference data sets used by us to experiment with contrasting data capture techniques and identify relevant features for our problem.

Dataset 1: *Keystroke Biometrics Ongoing Competition (KBOC)* [1]

- Data is unique to each user (first and last name)
- The data set contains 300 users, each with 4 labeled genuine samples and 20 unlabeled query samples.
- Considers variable length password and additional features such as sequence and usage of Modifier keys (Shift key, CapsLock key etc.).
- This is not ideal for our task because there arent many impostor data samples. Our primary hope to construct the adversarial attacks is by using data of a lot of users typing the same password. Therefore, we have focused mainly on the second data set.

Dataset 2: *Keystroke Dynamics - Benchmark Data Set (CMU)* [10]

- Unique password chosen (.tie5Roanl) for all users to type in. Killourhy and Maxion [2] refer to suggestions that letting users choose their own passwords makes it easier to discriminate them.
- 51 users; record 400 instances over 8 sessions per user. Sessions had a gap of at least one day.
- Out of 31 timing features recorded namely key-hold time, keyup-keydown times, keyup-keyup times for each keystroke in the password.
- Incorporates the modifier key timings with the associated password character key.

In order to use the abundant data set of CMU [10] data set, we transformed it to the format used by Monaco J. V. [1] (sequence of keystroke, action and timing information) using Pandas. We ran initial results on this transformed data set to check performance but the code had some specific nuances suitable for dataset 1 and did not incorporate some of the variables we wanted to experiment with. So, eventually we used the techniques of these classifiers and implemented them in python to get a handle on control parameters and variables needed.

We experimented with some preprocessing techniques to gauge impact on our otherwise best performing classifier (Manhattan- section IV.A.1):

- Skipping features: Results slightly better but with removing samples beyond 1 s.d. which is not very practical.
- Feature selection: No definite bias noticed for particular keystroke timing
- Shuffling user data: Results not too different

IV. Methods and Results

A. Classifiers

The classifiers for fraudulent attempt detection were implemented in Python similar to Monacos implementation [3] for KBOC.

- **Manhattan**: (or L1 norm) The taxicab distance, between two vectors p and q, in an n-dimensional real vector space with fixed Cartesian coordinate system, is the sum of the lengths of the projections of the line segment between the points onto the coordinate axes [4].

$$d_1(p,q) = \|p-q\|_1 = \textstyle\sum_{i=1}^{n} |p_i - q_i| \tag{1}$$

  Manhattan distance is calculated between the input vector and the mean timing feature vector. So, the model training is essentially computing the mean feature vector.
- **One Class SVM**: Adopted scikit-learn librarys implementation[5] which essentially calls libsvm [6]. It is an unsupervised model that learns a separating hyperplane between the origin and feature vector points [1]. We use parameter setting by Monaco [1] with = 0:5 (fraction of training errors (samples that lie outside the separating plane), radial basis function (RBF) kernel, and RBF kernel parameter = 0:9.
- **Autoencoder**: The implementation in tensorflow adopted here is a neural network which consists of same input and output layer with one hidden layer. The weight matrix (W) and bias vector(b_h) along with tanh function leads us to the hidden layer as follows:

$$h = f(x) = tanh(W^T x + b_h) \tag{2}$$

  The output layer with bias vector (b_y) is given by:

$$y = f(x) = tanh(W^T h + b_y) \tag{3}$$

  Parameters are determined by back propagating gradients from squared error loss function. The objective function of this basic autoencoder is:

$$J(\theta) = \textstyle\sum_n L(x, g(f(x))) \tag{4}$$

  As used by Monaco [1], bias parameters are initialized to 0 and weights are initialized from a random uniform distribution. Models are trained for 5000 epochs using gradient descent with a learning rate of 0.5. During testing, the score of a query sample is given by the negative reconstruction error, $-L(x;y)$. We use a single hidden layer of dimension 5.
- **Variational Autoencoder (VarAE)**: We wanted to use this technique to test the results but have used the implementation by Monaco [1]: *The variational autoencoder is probabilistic autoencoder with continuous latent variables. Parameters are learned efficiently by backpropagation through a reparametrization trick, which allows the gradient of the loss function to propagate through the sampling process. The objective function of the variational autoencoder is composed of both a reconstruction loss and a latent loss. It uses variational autoencoder with 2 hidden layers of dimension 5, spherical Gaussian latent space of dimension 3, and softplus nonlinearities between layers. Parameters are learned by Adam, a stochastic gradient-based optimization algorithm [7], using a learning rate of 0.001, mini-batch size of 2, and 700 epochs. Query sample scores are given by the negative reconstruction loss, which is the negative log probability of the input given the reconstructed latent distribution.*

TABLE I
Classification of user group and EER per user group

Users	Great	OK	Bad	Total
Total count	11	17	23	51
EER	0.019	0.061	0.179	0.113

TABLE II
Average Error rate per user group

Users	Manhattan	SVM	AutoEncoder	Var AE
Great	0.070	0.090	0.091	0.065
OK	0.084	0.096	0.096	0.100
Bad	0.134	0.136	0.149	0.139

Results: An ideal system would always accept the genuine input, and reject an incorrect input. This means there are two kind of errors possible: rejecting a correct input - False Reject Rate (FRR), and accepting a wrong input - False Accept Rate (FAR). Clearly, there is a tradeoff between these as can be seen from the following image:

To measure the performance of a system, the keystroke dynamics literature focuses on Equal error rate (ERR) which is the value at which both FAR and FRR are equal, with lower ERR, representing smaller overlapping regions and better classifiers.

We fit a subset of the data for training the classifiers (200 genuine, 200 imposters users). Then compute the score values for an equal number of labelled samples (200 genuine; 500 impostor users). The score from each sample enables us to compute the ROC (using sklearn) which gives us the ERR scores, and FAR and FRR curves. Based on the EER scores, we divided users into: Great (¡ 0.03), Ok (¡ 0.10), and Bad (¿ 0.10)

B. Adversarial Attackers

Methods: Here our objective was to use data from other users to artificially generate a timing vector that is used as input to the classifier. In all the attacks, we took as input 400 timing vectors of each of the 50 users besides the one we were targeting. Thus, in total, we had 20,000 samples of other users typing the same password as the target user. Next, we generated artificial vectors, by generating each required feature based on these samples. Once we generated an artificial timing vector, we tested the target users classifier on that.

The simplest possible attack was to generate each feature by averaging over all data points. This generated a single attack vector per target user. A more refined approach, as suggested by Dr.Bahman Bahmani, was to use k-means on the input of 20,000 timing vectors. This generated k-clusters, each of which could serve as an attack vector for the model. We ran this attack with k = 8, 16, 32, 64 and 128. In general, the nature of the attack does not allow for the attacker to have an unlimited number of tries, so smaller values of k were clearly more crucial.

Since the sum of averages, is the average of sums, the invariant of the 2nd feature being the sum of the first and third feature was maintained in the average attack. But at first, it was somewhat surprising for us to note that this invariant was maintained in the clusters computed by kmeans as well.

Results: In the graphs below, we show the number of compromised users per attack scenario for each of the classifiers. The average attack was surprisingly successful (almost 50% users were compromised), and for the kmeans attack, 8 clusters already beat the defences for most of the users. Also, notice that Manhattan and Variational Autoencoder seem to perform the best among the classifiers (lowest number of users that were compromised.)

Fig. 1. Number of users attack for which succeeded

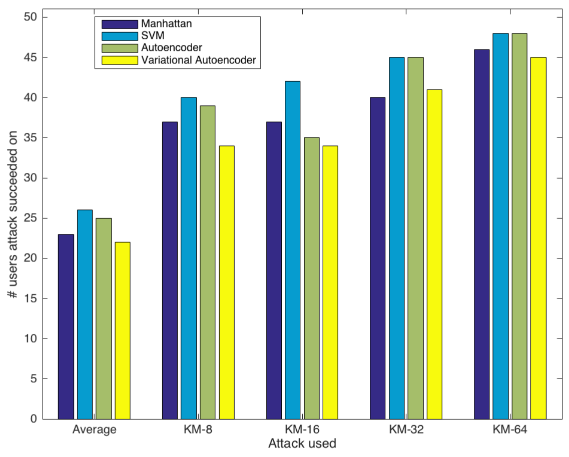

TABLE III
TOTAL NUMBER OF USERS K-MEANS ATTACKER FAILED TO BREAK FOR DIFFERENT CLUSTER SIZES

Classifiers	8 Clusters	16 Clusters	32 Clusters	64 Clusters
Man	14	14	11	5
SVM	12	9	6	3
AE	12	16	6	3
VarAE	17	15	10	6

C. Defences

We attempted many different defence schemes. Since the variational autoencoder took so much longer to train, and Manhattan was generally nearly as good in its results, here we mainly present the defence improvements with respect to the Manhattan and SVM classifiers.

TABLE IV
SPLIT UP OF USERS FAILED TO BREAK PER USER GROUP BY 8-CLUSTER K-MEANS ATTACKER

Classifier	Great	OK	Bad	Total
Man	4	5	5	14
SVM	3	4	5	12
AE	3	5	4	12
VarAE	3	6	8	17

TABLE V
SPLIT UP OF USERS FAILED TO BREAK PER USER GROUP BY 16-CLUSTER K-MEANS ATTACKER

Classifier	Great	OK	Bad	Total
Man	4	4	6	14
SVM	2	3	4	9
AE	6	3	7	16
VarAE	6	3	6	15

TABLE VI
SPLIT UP OF USERS FAILED TO BREAK PER USER GROUP BY 32-CLUSTER K-MEANS ATTACKER

Classifier	Great	OK	Bad	Total
Man	3	4	4	11
SVM	2	2	2	6
AE	2	3	1	6
VarAE	2	5	3	10

TABLE VII
SPLIT UP OF USERS FAILED TO BREAK PER USER GROUP BY 64-CLUSTER K-MEANS ATTACKER

Classifier	Great	OK	Bad	Total
Man	2	2	1	5
SVM	1	2	0	3
AE	0	2	1	3
VarAE	1	3	2	6

Initially, we tried to reduce the variability in the input data used by the classifier. It seemed possible to us that some users may never have got used to typing a password that wasnt theirs. As a first attempt, we just tried to get rid of the first few times the user typed the password in each session. But this did not have any noticeable improvements in the performance of the classifiers.

So instead, we decided to filter the inputs used for fitting the classifier. For each feature, we computed the statistics for all the genuine samples. And then we replaced values ¿ 1.5 Median absolute deviations away from the median of that feature with the median value (Note: We couldnt just throw away these values because that would have messed up the dimensions of the feature vectors). This resulted in almost twice the number of users being more resistant to the attacks (figure below) - but in the larger picture, from the 51 users, most users could still be compromised.

For practical purposes, perhaps the most robust defensive system would be to have more flexible threshold values. As described above, the threshold value was set to EER in order

Fig. 2. Filtered and Non-Filtered results with Manhattan classifier

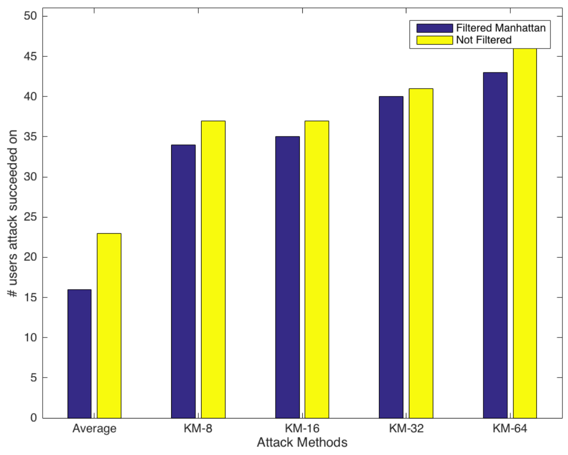

TABLE VIII
MEDIAN USED AS THRESHOLD WITH MANHATTAN CLASSIFIER AND K-MEANS ATTACKER WITH 32 AND 64 CLUSTERS

clusters	Great	OK	Bad	Total
32	8	15	12	35
64	8	14	8	30

to minimize the total errors. But if we only want to minimize the False Acceptance Rate, then we could have a much higher threshold value. Here, we set it to the median of the genuine samples:

The clear downside is that this considerably increases the False Rejection Rate, thus approximately half the time, a valid user will be flagged as a possible impostor as well. A possible way around it is to use such a conservative threshold only in special cases - for instance, when the user attempts a login from an unknown device. Then other secondary methods, as security questions could be used to confirm the identity of the user.

V. CONCLUSIONS

It is surprising how easily most of the user's defences could be compromised. This might make it appear that the keystroke dynamics layer does not add any additional security to the system. But if we use a more flexible threshold defence - as described in the last section above - then this might still add very useful security to most security critical internet applications.

The results also lead to the question - is it just a natural pattern that most users passwords werent strong enough, or perhaps, a lot of the users just never got used to typing in this particular password? To make more general conclusions, it should be very useful to run these tests on other datasets - especially - those that might have more 'natural' passwords, like the user's name.

REFERENCES

[1] John V. Monaco: Robust Keystroke Biometric Anomaly Detection, 2016
[2] Keystroke Biometrics Ongoing Competition (KBOC): https://sites.google.com/site/btas16kboc/home
[3] Monaco code base https://github.com/vmonaco/kboc
[4] https://en.wikipedia.org/wiki/Taxicab_geometry
[5] Fabian Pedregosa, Gal Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. The Journal of Machine Learning Research, 12:28252830, 2011.
[6] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. ACM Transactions on Intelligent Systems and Technology (TIST), 2(3):27, 2011.
[7] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980, 2014.
[8] https://nottmagazine.com/2015/08/31/infographic-how-keystroke-dynamics-tracks-you/
[9] A.Serwadda, V.V Phoha and A.Kiremire, Using Global Knowledge of Users Typing Traits to Attack Keystroke Biometrics Templates, in Proceedings of the Thirteenth ACM Multimedia Workshop, New York, 2011, pp. 51-60.
[10] http://www.cs.cmu.edu/˜keystroke/