Hindawi
Security and Communication Networks
Volume 2018, Article ID 9327215, 24 pages
https://doi.org/10.1155/2018/9327215

Research Article

Deep Learning Approaches for Predictive Masquerade Detection

Wisam Elmasry,[1] **Akhan Akbulut**,[2] **and Abdul Halim Zaim**[1]

[1]*Department of Computer Engineering, Istanbul Commerce University, Istanbul, Turkey*
[2]*Department of Computer Engineering, Istanbul Kultur University, Istanbul, Turkey*

Correspondence should be addressed to Wisam Elmasry; wisam.elmasry@istanbulticaret.edu.tr

Received 21 March 2018; Accepted 24 June 2018; Published 1 August 2018

Academic Editor: Mamoun Alazab

In computer security, masquerade detection is a special type of intrusion detection problem. Effective and early intrusion detection is a crucial factor for computer security. Although considerable work has been focused on masquerade detection for more than a decade, achieving a high level of accuracy and a comparatively low false alarm rate is still a big challenge. In this paper, we present a comprehensive empirical study in the area of anomaly-based masquerade detection using three deep learning models, namely, Deep Neural Networks (DNN), Long Short-Term Memory Recurrent Neural Networks (LSTM-RNN), and Convolutional Neural Networks (CNN). In order to surpass previous studies on this subject, we used three UNIX command line-based datasets, with six variant data configurations implemented from them. Furthermore, static and dynamic masquerade detection approaches were utilized in this study. In a static approach, DNN and LSTM-RNN models are used along with a Particle Swarm Optimization-based algorithm for their hyperparameters selection. On the other hand, a CNN model is employed in a dynamic approach. Moreover, twelve well-known evaluation metrics are used to assess model performance in each of the data configurations. Finally, intensive quantitative and ROC curves analyses of results are provided at the end of this paper. The results not only show that deep learning models outperform all traditional machine learning methods in the literature but also prove their ability to enhance masquerade detection on the used datasets significantly.

1. Introduction

In computer security domain, a masquerader is defined as an intruder seeking to mimic a genuine client. A masquerade attack takes place when a masquerader gets unauthorized access to a legitimate user's information by using his legitimate access credentials. These attacks are considered being among the most serious threats to computer security. The most effective way to prevent such attacks is using intrusion detection systems (IDSs) which can provide monitoring for all users and search for any abnormal conducts [1].

Computer security design incorporates with two common approaches of IDSs: signature-based detection and anomaly-based detection. Signature-based detection or also called misuse detection is valuable to use when the masquerade attack signature is already known. Alternatively, anomaly-based detection can be used for either known or unknown masquerade attacks. This advantage makes anomaly-based detection approach popular and a vast amount of prior studies has been published on this topic in the last decade [2]. The main idea behind anomaly-based detection approach is profiling the user behavior with collecting a variety of information about each user and then using this information to create a profile for each user depending on some characteristics. When the system is used, a security check is occurring to compare the recent activities done by the user with the original profile. If the user behavior deviates from the normal existing profile, then the session is classified to be as a possible masquerade attack. There are many anomaly-based detection techniques that are used, but among them, machine learning methods are the most commonly used approaches due to their ability to learn from data and then distinguish between normal and malicious users [3].

Despite the popularity of using traditional (shallow) machine learning methods for classification tasks, these have many deficiencies that need to be addressed, such as the perspective of full features representation, the complexity of the

problem, and limitation to static classification applications [4]. In 2006, a new concept of representation learning, based on Artificial Neural Network, called deep learning has been put forward. Deep learning is considered as a class of machine learning techniques that has, in hierarchical architectures, many layers of information processing stages for pattern recognition or classification. Rather than overcoming the former deficiencies of shallow machine learning methods, it achieves recently great success in many research fields. The main advantages of deep learning can be summarized as its practicability, having the ability to unsupervised feature learning or extraction from datasets, and having strong self-learning capability [5]. There are four typical models of deep learning, namely, Autoencoder (AE), Deep Belief Networks (DBN), Convolutional Neural Networks (CNN), and Recurrent Neural Networks (RNN). Because of deep learning success and stability, it has been actively and continually used in a wide range of applications nowadays such as computer vision, natural language processing, and intrusion detection systems [4, 5].

To our knowledge, neither of the previous studies on the area of the masquerade detection used deep learning to utilize its great capabilities and various learning models. The aim and contribution of this research are sixfold as follows: (i) we performed a comprehensive empirical study which investigates the effectiveness of three binary classification deep learning models to detect masqueraders' attacks; (ii) the first study uses three famous UNIX command line datasets with their different (six) data configurations and compares between them; (iii) we proposed a Particle Swarm Optimization-based algorithm for DNN hyperparameters selection; (iv) we carried out our experiments on all data configurations using both static and dynamic masquerade detection approaches; (v) we assessed the performance of the used deep learning models using twelve well-known evaluation metrics, Wilcoxon and Friedman statistical tests, and ROC analysis; (vi) we made comparisons between deep learning models' results and the best results of the traditional machine learning methods that have been published in the literature in the field of masquerade detection.

The rest of this paper is organized as follows: Section 2 reviews the related work that has been published previously in the area of masquerade detection using traditional machine learning methods and UNIX command line datasets. Then Section 3 describes UNIX command line datasets and their data configurations in detail. Section 4 presents a Particle Swarm Optimization-based algorithm to select hyperparameters of Deep Neural Networks (DNN). Section 5 shows how our experiments are established and what deep learning models are used, and Section 6 presents which evaluation metrics are used as well as it analyzes the gained experimental results. Finally, Section 7 presents our conclusions and possible future work.

2. Related Work

Masquerade detection has been actively researched in the last decade due to its significance and vulnerability to the computer security area. For the sake of brevity and restriction of the scope of this study, we have principally focused on anomaly-based masquerade detection using machine learning approaches and well-known UNIX command line-based datasets in the literature.

It was firstly introduced by Schonlau et al. [7] when they proposed a UNIX command line-based dataset called SEA. They also utilized various statistical methods on SEA data configuration and compared the results. In short time, SEA dataset becomes very popular in the field of anomaly-based masquerade detection techniques. T. Okamoto et al. [8] presented an immunity-based Hidden Markov Model on SEA data configuration and they got 60% Hit and 1% False Alarm Rate (FAR). Naive Bayes is a famous classifier that is working well with text classification tasks. It was first applied on SEA data configuration by Roy A. Maxion and Tahlia N. Townsend in 2002 [9] with two models, one with updating users profile (Hit=61.5%, FAR=1.3%) and the other with no-updating (Hit=66.2%, FAR=4.6%). Moreover, they proposed a new data configuration from SEA dataset named SEA 1v49 and also tested Naive Bayes classifier with updating on SEA 1v49 data configuration and they had 62.8% Hit and 4.6% FAR. K. Wang et al. in [10] implemented on SEA data configuration a Naive Bayes classifier (Hit=70%, FAR=2%) and One-Class Support Vector Machine (OCSVM) model (Hit=70%, FAR=4%). In the study [11], K. H. Yung presented in his work a Naive Bayes classifier with updating and feedback which has applied to SEA data configuration (Hit=76%, FAR=2%). He developed his previous work and proposed a self-consistent Naive Bayes model with updating on SEA data configuration in 2004 [12]. He had better results and increased Hit to 79%, but FAR is still 2%.

Support Vector Machine (SVM) is also a well-known machine learning method that is used for both classification and regression. Chen and Aritsugi introduced a SVM-based method for masquerade detection with online updating using Eigen Cooccurrence Matrix which is applied to SEA data configuration [13]. They tested their proposed method for One-Class (Hit=62.77%, FAR=6%) as well as for Two-Class (Hit=72.24%, FAR=3%) classification models. In 2006, Z. Li et al. extracted user behavior's principle features from Correlation Eigen Matrix using Principle Component Analysis (PCA), then they fed these features to SVM-based masquerade detection system on SEA data configuration [14]. They got a very good result with Hit=82.6% and FAR=3%. H. S. Kim and S. D. Cha performed an empirical study in the field of masquerade detection using SVM classifier with a voting engine [15]. They tested their SVM classifier on two UNIX command line-based datasets, namely, SEA dataset and Greenberg dataset [16] which latter is proposed by Greenberg in 1988. For SEA dataset, they applied their SVM classifier on two different data configurations, namely, SEA data configuration (Hit=80.1%, FAR=9.7%) and SEA 1v49 data configuration (Hit=94.8%, FAR=0%). In addition to that, they applied their SVM classifier on two different data configurations for Greenberg dataset, namely, Greenberg Truncated and Greenberg Enriched data configurations which are proposed by Maxion [17]. For Greenberg Truncated data configuration they had Hit=71.1% and FAR=6%; meanwhile, they had Hit=87.3% and FAR=6.4% for Greenberg

Table 1: Best results of the related works.

Model	Dataset	Configuration	Hit (%)	FAR (%)
HMM	SEA	SEA	60	1
Naive Bayes	SEA	SEA	79	2
		SEA 1v49	62.8	4.6
	Greenberg	Greenberg Truncated	70.9	4.7
		Greenberg Enriched	82.1	5.7
Conditional Naive Bayes	SEA	SEA	84	8.8
		SEA 1v49	90.7	1
	Greenberg	Greenberg Enriched	84.13	9.4
	PU	PU Enriched	84	8
SVM	SEA	SEA	82.6	3
		SEA 1v49	94.8	0
	Greenberg	Greenberg Truncated	71.1	6
		Greenberg Enriched	87.3	6.4
	PU	PU Enriched	60	2
Tree-based	PU	PU Enriched	85	10

Table 2: Datasets and their characteristics.

Dataset Name	Hosts Platform	No. of Users	Audit Format	Enriched?	Contaminated?	Sessions?	Real Masquerades?	Year
SEA	Unix	50	Unix Commands	No	Yes	No	No	2001
Greenberg	Unix	168	Unix Commands	Yes	No	Yes	No	1988
PU	Unix	8	Unix Commands	Yes	No	Yes	No	1997

Enriched data configuration. In 2007, Yang et al. presented a One-Class SVM with string kernel classifier to detect masquerade attacks [18]. They tested their classifier on two UNIX command line-based datasets, namely, SEA dataset and PU dataset [19] which latter is proposed by Lane and Brodley in 1997. For SEA dataset, they applied their model on SEA data configuration (Hit=62%, FAR=1.5%) and for PU dataset they applied their model on PU Enriched data configuration (Hit=60%, FAR=2%) which is proposed [19].

In the study [17], a Naive Bayes model with updating users profile is introduced in 2003 on both Greenberg Truncated and Greenberg Enriched data configurations, whereas Greenberg Truncated data configuration gave a Hit=70.9% and a FAR=4.7%, and Greenberg Enriched data configuration gave a Hit=82.1% and a FAR=5.7%. Gebski and Wong [20] presented a tree-based model for masquerade detection on PU Enriched data configuration (Hit=85%, FAR=10%). REDDY et al. proposed a conditional Naive Bayes classifier to detect masquerades [21]. They tested their classifier on three different UNIX command line-based datasets, namely, SEA, Greenberg, and PU datasets. For SEA dataset, they applied their classifier on two data configurations, namely, SEA data configuration (Hit=84%, FAR=8.8%) and SEA 1v49 data configuration (Hit=90.7%, FAR=1%). For Greenberg dataset, they applied their classifier on Greenberg Enriched data configuration (Hit=84.13%, FAR=9.4%). Finally, they tested their classifier on PU Enriched data configuration and they got a Hit=84% and a FAR=8%. Table 1 presents a summarization of the best results of the previous works above in terms of Hit percentage for each dataset. As we can notice from Table 1, developing a masquerade detection models for higher Accuracy and Hit as well as lower FAR values is still a big challenge.

3. Datasets and Configurations

This section describes the datasets that we used in our study, data configurations and the methodology of training and testing, as well. Indeed, there are various mechanisms that could be used to collect information about each user to model his behavior and then build his normal profile such as user command lines history, graphical user interface (GUI), user file system navigation, and system calls at the operating system level. In this paper, we selected three datasets based on UNIX command line history of users, namely, SEA, Greenberg, and PU. Rather than being free and publicly available on Internet, they are the most commonly used datasets in anomaly-based masquerade detection area, so our results will be easily compared to previous ones. Table 2 shows datasets and their characteristics.

3.1. SEA Dataset. Recently published papers that focused on masquerade detection area used this dataset. SEA (Schonlau Et Al.) is a free UNIX command line-based dataset [7]. They used UNIX acct audit tool to collect commands from 50 different users for several months. SEA dataset contains a set of 15000 commands for every user and these commands contain only command names issued by that user. For each user, the set of 15000 commands is divided into 150 blocks each with 100 commands. The first 50 blocks for each user are considered genuine and used as a training set. The remaining 100 blocks of each user are considered as a test set. Some of the test blocks are contaminated randomly with data of other users; i.e., each user has varying masquerader blocks in his test set from 0 to 24 blocks. Two associated data configurations have been used with this dataset in the literature: SEA and SEA 1v49.

3.1.1. SEA. This data configuration is proposed in the study [7]. A separate classifier is built for each of the 50 users. We trained each classifier to build two profiles: one profile for self-behavior using the first 50 blocks of the particular user and the other profile for non-self-behavior using (49×50) training blocks of the other 49 users. The test set of each user will be the same as described in Section 3.1.

3.1.2. SEA 1v49. In this configuration, we followed the same methodology proposed in research [9]. A classifier is built for each user and trained only with the first 50 training blocks of its data. On the other hand, the test set for each user consists of the first 50 training blocks of each of the other 49 users resulting in 2450 masquerade blocks in addition to its original normal blocks which vary between 76 and 100 blocks.

3.2. Greenberg Dataset. This dataset has been proposed in [16] and widely used in previous works. It contains commands collected from 168 UNIX users that used *csh* shell. Users of this dataset are considered to be a member in one of the following four groups: novice programmers, experienced programmers, computer scientists, and nonprogrammers. This dataset is enriched; i.e., it has sessions for each user including information about start and end time of the session, working directory, command names, command parameters, command aliases, and an error flag. Two associated data configurations have been used with this dataset in the literature: Greenberg Truncated and Greenberg Enriched.

3.2.1. Greenberg Truncated. In this configuration, we followed the same methodology conducted by [17]. First, we extracted the truncated command lines from Greenberg dataset which contain only the command names. Next, from 168 users available in Greenberg dataset we selected randomly 50 users who have between 2000 and 5000 commands to act as normal users. Then, we divided commands of each of the 50 users into blocks each with 10 commands. The first 100 blocks of each user will be his training set, whereas the next 100 blocks will be used as a validation of self-behavior in his test set. After that, we randomly selected additional 25 users from the remaining 118 users to act as masqueraders. Then, for each of the 50 normal users, we selected randomly 30 blocks from masqueraders' data and input them at random positions in his test set which results in a total of 130 blocks for testing.

3.2.2. Greenberg Enriched. It has the same methodology explained in Greenberg Truncated but with only one difference that for this data configuration we extracted only the enriched command lines from Greenberg dataset. Enriched command line means a concatenation of command name and command parameters entered by the user together with any alias employed. As for Greenberg Truncated data configuration described above, Greenberg Enriched data configuration has for each of the 50 normal users 100 blocks for training and 130 blocks for testing.

3.3. PU Dataset. Purdue University (PU) dataset has been proposed in [19]. It contains sanitized commands collected from 8 different users at Purdue University over the course of up to 2 years. This dataset is enriched which means that it contains, in addition to command names, command parameters, flags, and shell meta-characters. Furthermore, this dataset has sessions for each of the 8 users. In addition to that, data of each user is processed into a token stream. Token here means either command name or command parameter. Two associated data configurations have been used with this dataset in the literature: PU Truncated and PU Enriched.

3.3.1. PU Truncated. For this configuration, we followed the same methodology used in [19]. First, we extracted only the truncated tokens from PU dataset, i.e., the tokens that contain only command names. Next, for each of the 8 users available in PU dataset, we divided his data into blocks each of 10 tokens. Then, the first 150 blocks of each user will be considered as his training set. After that, the next 50 blocks for each user will be used as a validation of self-behavior in his test set. To simulate masquerade activities, we added, for each user, other seven users' testing data (7×50) which results in a total of 400 blocks of testing for each of the 8 users.

3.3.2. PU Enriched. It has the same methodology explained in PU Truncated, but with only one difference that, for PU Enriched data configuration, we extracted here only the enriched tokens, i.e., all tokens from PU dataset. As for PU Truncated data configuration described in Section 3.3.1, PU Enriched data configuration has for each of the 8 users 150 blocks for training and 400 blocks for testing. Table 3 summarizes all details about data configurations.

4. DNN Hyperparameters Selection

In this section, we will present a Particle Swarm Optimization-based algorithm to select the hyperparameters of Deep Neural Networks (DNN). This algorithm will help us to proceed in our experiments to construct DNN for masquerades detection as will be explained in Section 5.1. DNN is a multilayer Artificial Neural Network with many hidden layers. The weights of DNN are fully connected; i.e., every neuron at any particular layer is connected to all neurons of the higher-order layer that is located adjacently

Table 3: The structure of the used data configurations.

Characteristics		Data Configurations					
		SEA	SEA 1v49	Greenberg Truncated	Greenberg Enriched	PU Truncated	PU Enriched
Number of users		50	50	50	50	8	8
Block Size		100	100	10	10	10	10
Number of blocks for every user	Training set	2500	50	100	100	150	150
	Test set	100	2526~2550	130	130	400	400
	Total	2600	2576~2600	230	230	550	550
Number of blocks for all users	Training set	125000	2500	5000	5000	1200	1200
	Test set	5000	127269	6500	6500	3200	3200
	Total	130000	129769	11500	11500	4400	4400
Distribution of the training set	Normal	2500	2500	5000	5000	1200	1200
	Masquerader	122500	0	0	0	0	0
	Total	125000	2500	5000	5000	1200	1200
Distribution of the test set	Normal	4769	4769	5000	5000	400	400
	Masquerader	231	122500	1500	1500	2800	2800
	Total	5000	127269	6500	6500	3200	3200

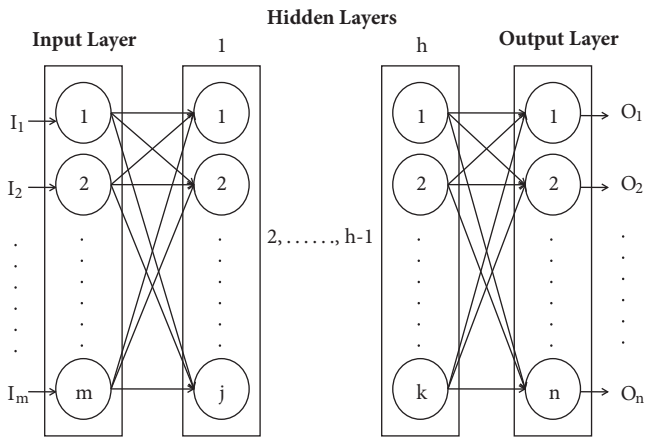

Figure 1: The basic structure of a typical DNN.

to that particular layer [4]. The information in DNN is propagated in a feed-forward manner, that is, from inputs to outputs via hidden layers. Figure 1 depicts the basic structure of a typical DNN.

DNNs are widely used in various machine learning tasks. In addition to that, they have proved their ability to surpass most of the machine learning techniques in terms of performance [22]. However, the performance of any DNN relies on the selection of the values of its hyperparameters. DNN hyperparameters are defined as a set of critical parameters that control the architecture, behavior, and performance of that DNN in the underlying machine learning task. Indeed, there are two kinds of such hyperparameters: global parameters and layer-based parameters. The global parameters are those that defined the general behavior of DNN such as learning rate, epochs number, batch size, number of layers, and the used optimizer. On the other hand, layer-based parameters values are dependent on each layer in DNN. Examples of layer-based parameters are, but not limited to, type of layer, weight initialization method, activation function, and a number of neurons.

The problem is that these hyperparameters are varying from task to task and they must be set before the training process. One familiar solution to overcome this problem is to find an expert who is conversant with the underlying machine learning task to tune precisely the DNN hyperparameters. Unfortunately, the existence of such expert is not available in all cases. Another possible solution is to adjust these hyperparameters manually in a trial-and-error manner. This can be handled by searching the space of hyperparameters by executing either grid search or random search [23, 24]. A grid search is performed upon defined ranges of hyperparameters where those ranges are identified previously depending on a prior knowledge of the underlying task. After that, the user picks up values of hyperparameters from the predefined ranges consecutively and tests the performance of DNN on the training set. When all possible combination of hyperparameters values is tested, the best combination is selected to configure DNN and test it on the test set. Random search is similar to grid search, but instead of picking up hyperparameters values in a methodical manner, the user selects hyperparameters values from those predefined ranges randomly. In 2012, Snoek et al. have proposed a hyperparameters selection method based on Bayesian optimization [25]. In this method, the user improves his knowledge of selecting hyperparameters by using the information gained from any given experiment to decide how to adjust the hyperparameters for the next experiment. Despite good results that have been obtained by the grid, random, and Bayesian optimization searches in some cases, in general, the complexity and large search

space of the DNN hyperparameters values make such manual algorithms infeasible and too exhausting searching process.

Evolutionary Algorithms (EAs) are metaheuristic algorithms which perform excellently for finding the global optima of a nonlinear function, especially when there are multiple local minima or maxima. EAs are considered as very promising algorithms for solving the problem of DNN parameterization automatically. In the literature, there are a lot of studies that have been proposed recently aiming at using EAs in optimizing DNN hyperparameters in order to gain a high accuracy value as much as possible. Genetic Algorithm (GA), which is one of the most famous EAs, has been used to optimize the network parameters and the Taguchi method is applied between the crossover and mutation operators including initial weights definition [26]. GAs also are used in the pretraining step prior to the supervised step based on a multiclass classification task [27]. Another approach using GA to reduce the training time has been presented in [28]. The GA is used to enhance Deep Neural Networks by evolving a neural network's weights [29]. An automated GA-based approach has been proposed in [30] that optimized DNN hyperparameters for malware classification tasks. Moreover, Particle Swarm Optimization is also one of the most well-known and popular EAs. Lorenzo et al. used PSO and proposed two approaches; the first is sequential and the second is parallel, to optimize hyperparameters of any DNN [31, 32]. Then, Nalepa and Lorenzo proved formally the convergence abilities of the former two approaches and tested them separately on a single workstation and a cluster of sequential and parallel approaches, respectively [33]. Finally, F. Ye proposed in 2017 an automatic PSO-based algorithm to select DNN hyperparameters in large scale and high dimensional data [34]. Thus, we decided to use PSO to enable us to select hyperparameters for DNN automatically. Then, in Section 5.1 we will explain how to adapt this algorithm for static classification experiments used in a masquerade detection scenario. Section 4.1 introduces a necessary and brief preface reviewing how standard PSO is working. Then, the rest of this section presents our proposed PSO-based algorithm to optimize DNN hyperparameters.

4.1. Particle Swarm Optimization. Particle Swarm Optimization (PSO) is a metaheuristic algorithm for optimizing nonlinear functions in continuous search space. It was proposed by Eberhart and Kennedy in 1995 [35]. PSO tries to mimic the social behavior of animals. The swarm concept is a set of many members which are called particles. The number of particles in the swarm is an integer value denoted by S and called swarm size. Every particle in the particular swarm has two vectors of N length, where N is the size of the problem defined variables (dimensions). The first vector is called position vector denoted by P that identifies the current position of that particle in the search space of the problem. Each position vector can be considered as a candidate solution of the problem. The second vector is called velocity vector denoted by V that determines both speed and direction of that particle in the search space of the problem at next iteration. During the execution of PSO, another two vectors at every iteration should be stored. The first is called personal best vector denoted by P^i_{best} which indicates the best position of the ith particle in the swarm that has been explored so far. Each particle in the swarm has its independent personal best vector from the other particles and it is updated at each iteration. The second vector is the global best vector denoted by G_{best} which indicates the best position that has been found over the swarm so far. There is a single global best vector for all particles in the swarm and it is updated at every iteration. It can be looked to personal best vector as the cognitive knowledge of the particle, whereas the global best vector represents the social knowledge of the swarm. Mathematically, for each particle i in the swarm S at each iteration t the velocity V and position P vectors are updated to next iteration $t+1$ according to (1) and (2), respectively.

$$V^i_{t+1} = WV^i_t + C_1 r_1(t)\left(P^i_{best} - P^i_t\right) + C_2 r_2(t)\left(G_{best} - P^i_t\right) \quad (1)$$

$$P^i_{t+1} = P^i_t + V^i_{t+1} \quad (2)$$

W is the inertia weight constant which controls the impact of the velocity of the particle at the current iteration on the next iteration, so the speed and direction of the particle are adjusted in order not to let the particle to get outside the search space of the problem. Meanwhile, C_1 and C_2 are constants and known as acceleration coefficients; r_1 and r_2 are random values uniformly distributed in $[0, 1]$. At the beginning of every iteration, new values of r_1 and r_2 are computed randomly and they are constants for all particles in the swarm at that iteration. The goal of using C_1, C_2, r_1, and r_2 constants is to scale both the cognitive knowledge of the particle and the social knowledge of the swarm on the velocity changes. So, the new position vectors of all particles will approach to the optimal solution of the problem accordingly. Figure 2 depicts the flowchart of the standard PSO.

In brief, the standard PSO works as follows: First, the user enters some required inputs like swarm size (S), dimensions of the particles (N), acceleration constants (C_1, C_2), inertia weight constant (W), fitness function (F) to score particle performance in the problem domain, and the maximum number of iterations (t_{max}). Next, PSO initializes position and velocity vectors with the specified dimensions for all particles in the swarm randomly. Then, PSO initializes the personal best vector for each particle in the swarm with the specified dimensions and sets them to very small value. Furthermore, PSO initializes the global best vector of the swarm with the specified dimensions and sets it to very small value. PSO computes the fitness score for each particle using the fitness function and updates the personal best vectors for all particles and the global best vector of the swarm. After that, PSO starts the first iteration by computing r_1 and r_2 randomly and then updates velocity and position vectors for each particle according to (1) and (2), respectively. In addition to that, PSO computes again the fitness score for each particle according to the given fitness function and updates the personal best vector for each particle if the fitness score of that particle at this iteration is bigger than the fitness

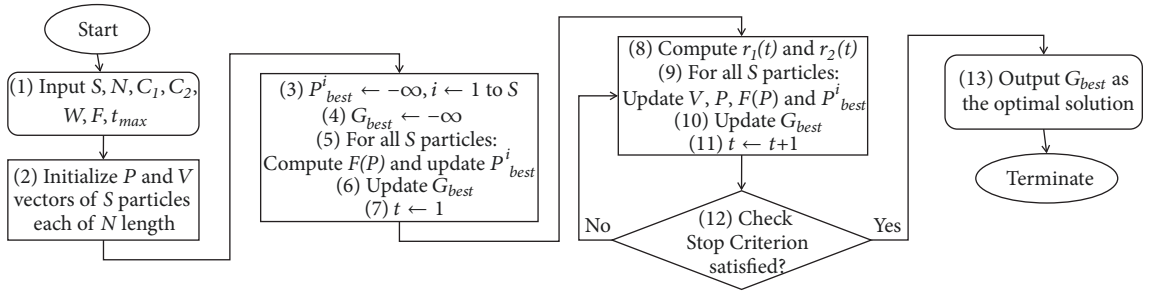

Figure 2: The flowchart of the standard PSO.

score of the personal best vector of that particle ($F(P^i_t) > F(P^i_{best})$). Also, PSO updates the global best vector of the swarm if any of the fitness score of the personal best vector of the particles is bigger than the fitness score of the global best vector of the swarm ($F(P^i_{best}) > F(G_{best})$, i=1 to S). Then, PSO checks the stop criterion and if one is satisfied, PSO will output the global best vector as the optimal solution and terminate. Else, PSO will proceed to the next iteration and repeat the same procedure described in the first iteration above until the stop criterion is reached.

The stop criterion is satisfied when either the training error is smaller than a predefined value ($\mathfrak{E}$) or the maximum number of iteration is reached. Finally, PSO performs better than GA in terms of simplicity and generality [36]. PSO is simpler than GA because it contains only one operator and easy to implement. Also, the generality of PSO means that PSO does not need any modifications to be applied to any optimization problem as well as it is faster to converge to the optimal solution which decreases the computations and saves the resources.

4.2. DNN Hyperparameters Selection Using PSO. The selection of the hyperparameters of DNN can be interpreted as an optimization task; hence the main objective is to minimize the loss function $L(M,T)$, where M is the DNN model and T is the training set. To achieve this goal, we selected PSO to be our optimization algorithm that outputs the vector of the optimized hyperparameters H that minimized the loss function L after constructed DNN model M which is tuned by the hyperparameters H and trained on the training set T. The fitness function of our PSO-based algorithm is a function F^*: $R^N \longrightarrow R$ that maps a real-valued vector of hyperparameters that has a length of N to a real-valued accuracy value of the trained DNN that is tuned by that hyperparameters vector and tested on the test set Z. In other words, our PSO-based algorithm finds the optimal hyperparameters vector among all possible combinations of hyperparameters, which yields to maximize the accuracy of the trained DNN on the test set. Furthermore, to ensure the generality of our PSO-based algorithm which means to be independent of the DNN that will be optimized and be adapted easily to any classification task using DNN, we will allow the user to select which hyperparameters want to use in his work. Therefore, the user is responsible for using our algorithm to define the number of the hyperparameters as well as the type and domain of each parameter. The domain of a parameter is the set of all possible values of that parameter. After that, our PSO-based algorithm will use a special built-in generator that depends on the number and domains of the defined parameters to initialize all the particles (hyperparameters vectors) in the swarm.

During the execution of the proposed algorithm and at each iteration, the validation process is involved in the proposed algorithm to validate the updated position and velocity vectors to be appropriate to the predefined ranges of parameters. Finally, in order to reduce computations and converge faster, two different stop conditions are checked simultaneously at the end of each iteration. The first occurs when the fitness score of the global best vector increased less than a threshold $\mathfrak{E}$ which is specified by the user. The aim of the former condition is to guarantee that the global best vector cannot be improved further, even if the maximum number of iterations is not reached yet. The second condition happens when the maximum number of iterations is carried out. Either the first or the second condition is satisfied, then the proposed algorithm outputs the global best vector as the optimal solution H and terminates the search process. Figure 3 shows the flowchart of our PSO-based DNN hyperparameters selection algorithm.

4.3. Algorithm Steps

Inputs: Number of hyperparameters (N), swarm size (S), acceleration constants (C_1, C_2), inertia constant (W), maximum value of velocity (V_{max}), minimum value of velocity (V_{min}), maximum number of iterations (t_{max}), evolution threshold ($\mathfrak{E}$), training set (T), and test set (Z).
Output: The optimal solution H.
Procedure:

Step 1. For k⟵1 to N

Let h^k be the k^{th} hyperparameter

If domain of h^k is continuous then

let B^k_{low} be the lower bound of h^k and B^k_{up} be the upper bound of h^k

let user enter the lower and upper bounds of a hyperparameter h^k

End of if

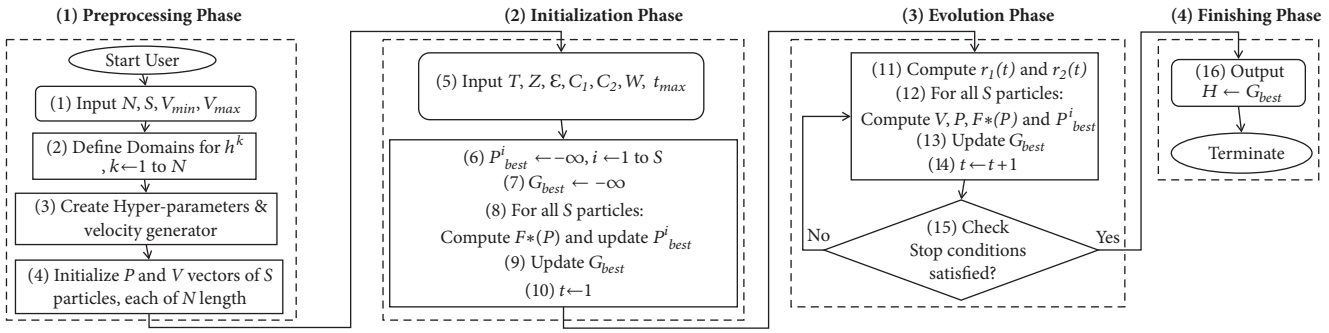

Figure 3: The flowchart of the proposed algorithm.

Else

Let Y^k be the set of all possible values of h^k

Let user enter all elements of the set Y^k

End of else

End of for

Step 2. Let F^* be the fitness function which constructs DNN tuned with the given hyperparameters, then trains DNN on T, and tests it on Z. Finally, F^* computes the accuracy of DNN as output.

Step 3. Let G_{best} be the global best vector of the swarm of length N.

Let GS be the best fitness score of the swarm

$GS \longleftarrow -\infty$

Step 4. For i$\longleftarrow$1 to S

Let P^i be the position vector of the ith particle of length N

Let V^i be the velocity vector of the ith particle of length N

Let P^i_{best} be the personal best vector of the ith particle of length N

Let PS_i be the fitness score of the personal best vector of the ith particle

For j$\longleftarrow$1 to N

If domain of h^j is continuous then select h^j uniformly distributed

$P[j] \longleftarrow \mathrm{U}(B^j_{low}, B^j_{up})$

End of if

Else

Select h^j randomly by P^i[j] $\longleftarrow$ RAND (Y^j)

End of else

$V^i[j] \longleftarrow \mathrm{U}(V_{min}, V_{max})$

End of for

$P^i_{best} \longleftarrow P^i$

Let FS_i be the fitness score of the ith particle

$FS_i \longleftarrow F^*(P^i)$

$PS_i \longleftarrow FS_i$

If $FS_i > GS$ then

$Gbest \longleftarrow P^i$

$GS \longleftarrow FS_i$

End of if

End of for

Step 5. Let GS_{prv} be the previous best fitness score of the swarm

$GS_{prv} \longleftarrow GS$

Let r_1 and r_2 be random values in PSO

Let t be the current iteration

For $t\longleftarrow$1 to t_{max}

$r_1 \longleftarrow U(0, 1)$

$r_2 \longleftarrow U(0, 1)$

For $i\longleftarrow$ 1 to S

Update V^i according to (1)

Update P^i according to (2)

$FS_i \longleftarrow F^*(P^i)$

If $FS_i > PS_i$ then

$sssP^i_{best} \longleftarrow P^i$

$PS_i \longleftarrow FS_i$

End of if

If $PS_i > GS$ then

$Gbest \longleftarrow P^i_{best}$

$GS \longleftarrow PS_i$

End of if

End of for

If $\|GS\text{-}\ GS_{prv}\| < \mathcal{E}$ then

go to Step 6

End of if

Table 4: PSO parameters recommended values or ranges.

Parameter	Value/Range
S	[5, 20]
V_{min}	0
V_{max}	1
C_1	2
C_2	2
W	[0.4, 0.9]
t_{max}	[30, 50]
$\mathcal{E}$	0.0001

$GS_{prv} \longleftarrow GS$
End of for

Step 6. Let H be the optimal hyperparameters vector

$H \longleftarrow Gbest$
Return H and Terminate

4.4. PSO Parameters. Selection of the value of PSO parameters (S, V_{max}, V_{min}, C_1, C_2, W, t_{max}, $\mathcal{E}$) is a very complex process. Fortunately, many empirical and theoretical previous studies have been published to solve this problem [37–40]. They introduced some recommended values of PSO parameters which can be taken. Table 4 shows every PSO parameter and the corresponding recommended value or range. Thus, for those parameters which have recommended ranges, we can select a value for each parameter from its range randomly and fix it as a constant during the execution of PSO.

5. Experimental Setup and Models

This section explains the methodology of performing our empirical experiments as well as the description of deep learning models which we used to detect masquerades. As mentioned in Section 3, we selected three UNIX command line-based datasets (SEA, Greenberg, PU). Each of these datasets is a collection of text files in which each text file represents a user. The text file of each user in the particular dataset contains a set of UNIX commands that are issued by that user. This reflects the fact that these datasets do not contain any real masqueraders. However, to simulate masqueraders and to use these datasets in masquerade detection, special data configurations must be implemented prior to proceeding in our experiments. According to Section 3 and its subsections, each dataset has its two different types of data configurations. Therefore, we obtained six data configurations that each one will be observed separately which yields, in the result, to six independent experiments for each model. Finally, masquerade detection can be applied to these data configurations by following two different main approaches, namely, static classification and dynamic classification. The two subsequent subsections present the difference between them as well as which deep learning models are exploited for each one.

5.1. Static Classification Approach. In the static classification approach, the classification task is carried out using a dataset of samples, which are represented by a set of static features [30]. These static features are defined according to the nature of the task where the classification will be applied. In addition to that, the dataset samples, or also called observations, are collected manually by some experts working in the field of that classification task. After that, these samples are split into two independent sets known as training and test sets to train and test the selected model, respectively. Static classification approach has pros and cons as well. Although it provides a faster and easier solution, it requires a ready-to-use dataset with static features. The existence of such dataset might not be available in some complex classification tasks. Hence, the attempt to create a dataset with static features will be a hard mission. In our work, we decided to utilize the existence of three famous UNIX command line-based datasets to implement six different data configurations. Each user in the particular data configuration has a specific number of blocks, which are represented by a set of static features. Indeed, these features are the user's UNIX commands, in charge of describing the behavior of that user and later helping the classifier to detect masquerades. We decided to use two well-known deep learning models, namely, Deep Neural Networks (DNN) and Recurrent Neural Networks (RNN) to accomplish the static masquerade detection task on the implemented six data configurations.

5.1.1. Deep Neural Networks. In Section 4, we explained in detail the DNN structure and the problem of the selection of its hyperparameters. We also proposed PSO-based algorithm to obtain the optimal hyperparameters vector that maximized the accuracy of the DNN on the given training and test sets. In this subsection, we describe how we utilized the proposed PSO-based algorithm and the DNN in static masquerade detection task using six of data configurations, which are SEA, SEA 1v49, Greenberg Truncated, Greenberg Enriched, PU Truncated, and PU Enriched. Every data configuration of them has its structure and a specific number of users as described in Section 3. So, we will have six separate DNN-experiments, and each experiment will be on one of the data configurations.

The methodology of our DNN-experiments consists of four consecutive stages, which are initialization, optimization, results extraction, and finishing stages. The first stage is to initialize all required operating parameters as well as to prepare the particular data configuration's files in which each file represents a user in that data configuration. The user file consists of the training set followed by the test set of that user. We set all PSO parameters for all DNN-experiments as follows: S=20, V_{min}=0, V_{max}= 1, C_1=C_2=2, W=0.9, t_{max}=30, and $\mathcal{E}$=10^{-4}. Then, the last step in the initialization stage is to define hyperparameters of the DNN and their domains. We used twelve different DNN hyperparameters (N=12). Table 5 shows each DNN hyperparameter and its corresponding defined domain. All the used hyperparameters are numerical except that Optimizer, Layer type, Initialization function, and Activation function hyperparameters are categorical. In this case, a list of all possible values is indexed to a sequenced-numbered range from 1 to the length of that list. Optimizer list includes elements: Adagrad, Nadam, Adam, Adamax,

Table 5: The used DNN hyperparameters and their domains.

Hyperparameter	Domain	Description
Learning rate	[0.01, 0.9]	Continuous
Momentum	[0.1, 0.9]	Continuous
Decay	[0.001, 0.01	Continuous
Dropout rate	[0.1, 0.9]	Continuous
Number of hidden layers	[1, 10]	Discrete with step=1
Numbers of neurons of hidden layers	[1, 100]	Discrete with step=1
Number of epochs	[5, 20]	Discrete with step=5
Batch size	[100, 1000]	Discrete with step=50
Optimizer	[1, 6]	Discrete with step=1
Initialization function	[1, 8]	Discrete with step=1
Layer type	[1, 2]	Discrete with step=1
Activation function	[1, 8]	Discrete with step=1

RMSprop, and SGD. Layer type list contains two elements, which are Dropout and Dense. Initialization function list includes elements: Zero, Normal, Lecun uniform, Uniform, Glorot uniform, Glorot normal, He uniform, and He normal. Finally, Activation list has eight elements, which are Linear, Softmax, ReLU, Sigmoid, Tanh, Hard Sigmoid, Softsign, and Softplus. It is worth mentioning that the elements of all categorical hyperparameters are defined in Keras implementation [30].

The optimization and results extraction stages will be performed once for each user in the particular data configuration; that is, they will be repeated for each user U_i, i=1,2,. . ., M, where M is the number of users in the particular data configuration D. The optimization stage starts by splitting the data of the user U_i into two independent sets T_i and Z_i, which are the training and test sets of the ith user, respectively. The splitting process followed the structure of the particular data configuration which is described in Section 3. All blocks of the training and test sets are converted from text to numeric values and then are normalized in $[0, 1]$. After that, we supplied these sets to the proposed PSO-based algorithm to find the optimized hyperparameters vector H_i for the ith user. In addition to that, we will save a copy of H_i values in a database, in order to save time and use them again in the RNN-experiment of that particular data configuration D, as will be presented in Section 5.1.2. The results extraction stage takes place when constructing the DNN that is tuned by H_i, trains the DNN on T_i, and tests the DNN on Z_i. The values of the classification outcomes, True Positive (TP_i), False Positive (FP_i), True Negative (TN_i), and False Negative (FN_i) for the ith user in the particular data configuration D, are extracted and saved for further processing later.

Then, the next user is observed and same procedure of optimization and results extraction stages is performed, till the last user in the particular data configuration D is reached. Finally, when all users in the particular data configuration are completed, the last stage (finishing stage) is executed. Finishing stage computes the summation of all obtained TPs of all users in the particular data configuration D denoted by TP. The same process will be applied also to the other outcomes, namely, FP, TN, and FN. Equations (3), (4), (5), and (6) express the formulas of TP, FP, TN, and FN, respectively.

$$TP = \sum_{i=1}^{M} TP_i \tag{3}$$

$$FP = \sum_{i=1}^{M} FP_i \tag{4}$$

$$TN = \sum_{i=1}^{M} TN_i \tag{5}$$

$$FN = \sum_{i=1}^{M} FN_i \tag{6}$$

The finishing stage will report and save these outcomes and end the DNN-experiment for the particular data configuration D. The former outcomes will be used to compute ten well-known evaluation metrics to assess the performance of the DNN on the particular data configuration D, as will be presented in Section 6. It is worth saying that the same procedure which is explained above will be done for each data configuration. Figure 4 depicts the flowchart of the methodology of the DNN-experiments.

5.1.2. Recurrent Neural Networks. The Recurrent Neural Network is a special type of the traditional feed-forward Artificial Neural Network. Unlike traditional ANN, in the RNN, each neuron in any of the hidden layers has additional connections from its output to itself (self-recurrent) as well as to other neurons of the same hidden layer. Therefore, the output of the RNN's hidden layer at any time step (t) is for the current inputs and the output of the hidden layer at the previous time step (t-1). In RNN, these directed cycles allow information to circulate in the network and make the hidden layers as the storage unit of the whole network [41]. The important characteristics of the RNN are the capability to have memory and generate periodical sequences.

Despite that, the conventional RNN structure which is described above has a serious problem especially when the

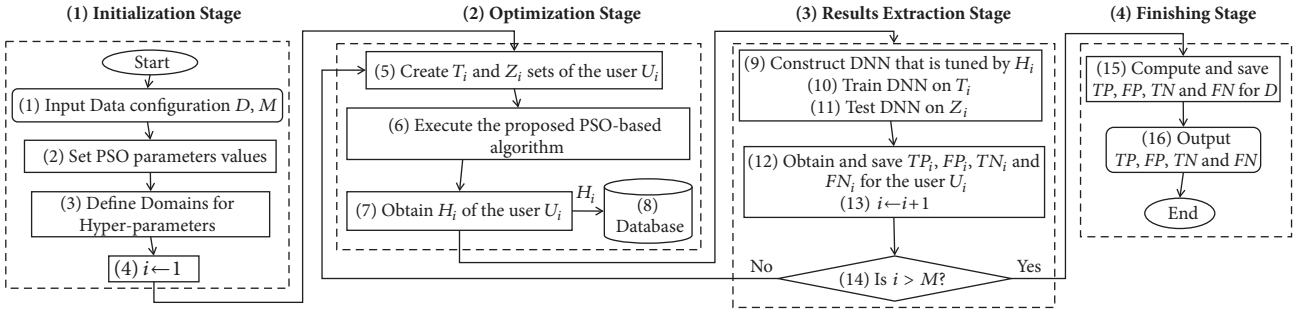

Figure 4: The flowchart of the DNN-experiments.

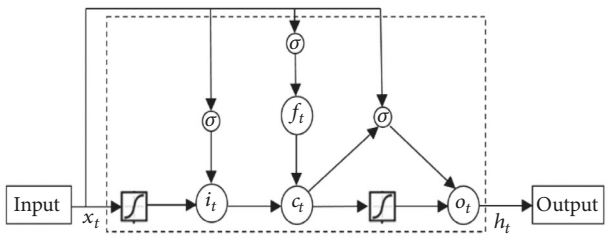

Figure 5: The structure of an LSTM cell [6].

RNN is trained using the back-propagation technique. The problem is known as gradient vanishing and exploding [42]. The gradient vanishing problem occurs when the gradient signal gets so small over the network which causes learning to become very slow or stop. On the other hand, the gradient exploding problem occurs when the gradient signal gets so large in which learning diverges. This problem of the conventional RNN limited the use of the RNN to be only in short-term memory tasks. To solve this problem, a new architecture of RNN is proposed by Hochreiter and Schmidhuber [43] known as Long Short-Term Memory (LSTM). LSTM uses a new structure called a memory cell that is composed of four parts, which are an input gate, a neuron with a self-recurrent connection, a forget gate, and the output gate. Meanwhile, the main goal of using a neuron with a self-recurrent connection is to record information; the aim of using three gates is to control the flow of information from or into the memory cell. The input gate decides if to allow the incoming information to enter into the memory cell or block it. Moreover, the forget gate controls if to pass the previous state of the memory cell to alter the current state of the memory cell or prevent it. Finally, the output gate determines if to pass the output of the memory cell or not. Figure 5 shows the structure of an LSTM memory cell. Rather than overcoming the problems of the conventional RNN, LSTM model also outperforms the conventional RNN in terms of performance especially in long-term memory tasks [5]. The LSTM-RNN model can be obtained by replacing every neuron in the hidden layers of the RNN to an LSTM memory cell [6].

In this study, we used the LSTM-RNN model to perform a static masquerade detection task on all data configurations. As mentioned in Section 5.1.1, there are six data configurations and each of them will be used in the separate experiment. So, we will have six separate LSTM-RNN-experiments; each experiment will be on one of the data configurations. The methodology of all of these experiments is the same and as follows: for the given data configuration D, we firstly prepared all the given data configuration's files by converting all blocks from text to numerical values and then normalizing them in $[0, 1]$. Next to that, for each user U_i in D, where i=1,2,. . ., M and M is the number of users in D, we did the following steps: we split the data of U_i into two independent sets T_i and Z_i, which are the training and test sets of the ith user in D, respectively. The splitting process followed the structure of the particular data configuration which is described in Section 3. After that, we retrieved the stored optimized hyperparameters vector of the ith user (H_i) from the database which is created in the previous DNN-experiments. Then, we constructed the RNN model that is tuned by H_i. In order to obtain the LSTM-RNN model, every neuron in any of the hidden layers is replaced to an LSTM memory cell. The constructed LSTM-RNN model is trained on T_i and then tested on Z_i. After the test process finished, we extracted and saved the outcomes: TP_i, FP_i, TN_i, and FN_i of the ith user in D. Then, we proceed to the next user in D to do the same previous steps until the last user in D is reached. After all users in D are completed, we computed the overall outcomes TP, FP, TN, and FN of the data configuration D by using (3), (4), (5), and (6), respectively. Figure 6 depicts the flowchart of the methodology of LSTM-RNN-experiments.

5.2. Dynamic Classification Approach. In contrast of static classification approach, dynamic classification approach does not need a ready-to-use dataset with static features [30]. It covenants directly with raw data sources such as text, image, video, sound, and signal files and extracts features from them dynamically. The models that use this approach try to learn and represent features in unsupervised manner. Then, these models train themselves using the extracted features to be able to classify unseen data. The deep learning models fit very well for this approach, because the main objectives of deep learning models are the strong ability of automatic feature extraction, and self-learning. Rather than that, dynamic classification models overcome the problem of the lake of datasets; it performs more efficient than the static classification models. Despite these advantages, dynamic classification approach has also drawbacks. Dynamic classification models are slower and take a long time to train, if compared with

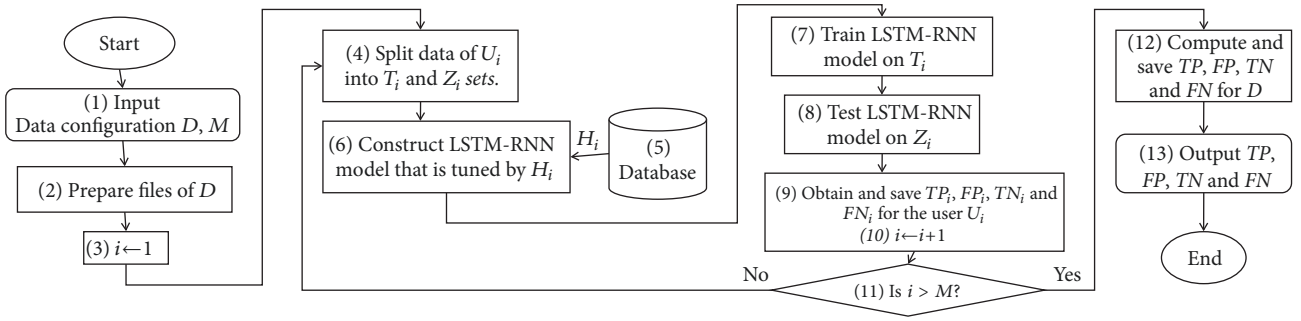

Figure 6: The flowchart of the LSTM-RNN-experiments.

static classification models, due to complex deep structure of these models as well as the huge amount of computations that are required to execute. Furthermore, dynamic classification models require a very large amount of input samples to gain high accuracy values.

In this research, we used six data configurations that are implemented from three textual datasets. In order to apply dynamic masquerade detection on these data configurations, we need a model that is able to extract features from the user's command text file dynamically and then classify the user into one of the two classes that will be either a normal user or a masquerader. Therefore, we deal with a text classification task. The text classification is defined as a task that assigns a piece of text (a word, a sentence, or even a document) to one or more classes according to its content. Indeed, there are three types of text classification, namely, sentence classification, sentiment analysis, and document categorization. In sentence classification, a given sentence should be assigned correctly to one of possible classes. Furthermore, sentiment analysis determines if a given sentence is a positive, negative, or neutral towards a specific subject. In contrast, document categorization deals with documents and determines which class from a given set of possible classes a document belongs to. According to the nature of dynamic classification as well as the functionality of text classification, deep learning models are the fittest among the other machine learning models for these types of classification due to their powerful capability of features learning.

A wide range of researches have been accomplished in the literature in the field of text classification using deep learning models. It was started by LeCun et al. in 1998 when they proposed a special topology of the Convolutional Neural Network (CNN) known as LeNet family and used it in text classification efficiently [44]. Then, various studies have been published to introduce text classification algorithms as well as the factors that impact the performance [45–47]. In the study [48], the CNN model is used for sentence classification task over a set of text dataset benchmarks. A single one-dimensional CNN is proposed to learn a region-based text embedding [49]. X. Zhang et al. introduced a novel character-based multidimensional CNN for text classification tasks with competitive results [50]. In the research [51], a new hierarchal approach called Hierarchal Deep Learning for Text classification (HDLTex) is proposed and three deep structures, which are DNN, RNN, and CNN, are used. A recurrent convolutional network model is introduced [52] for text classification and high results are obtained on documents-level datasets. A novel LSTM-based model is introduced and used for text classification with multitask learning framework [53]. The study [54] proposed a new model called hierarchal attention network for document classification and is tested on six large document-level datasets with good results. A character-level text representations approach is proposed and tested for text classification tasks using deep CNN [55]. As noticed, the CNN is the mostly used deep learning model for text classification tasks. So, we decided to use the CNN to perform dynamic masquerade detection on all data configurations. The following subsection reviews the CNN and explains the structure of the used CNN model and the methodology of our CNN-experiments.

5.2.1. Convolutional Neural Networks. The Convolutional Neural Network (CNN) is a deep learning model which is biological-inspired from the animal visual cortex. The CNN can be considered as a special type of the traditional feed-forward Artificial Neural Network. The major difference between ANN and CNN is that instead of the fully connected architecture of ANN, the individual neurons in CNN are connected to subregions of the input field. The neurons of the CNN are arranged in such a way they are tilled to cover the entire input field. The typical CNN consists of five main components, namely, an input layer, the convolutional layer, the pooling layer, the fully connected layer, and an output layer. The input layer is where the input data is entered into the CNN. The first convolutional layer in the CNN consists of individual neurons that each of them is connected to a small subset of the input field. The neurons in the next convolutional layers connect only to a subset of their preceding pooling layer's output. Moreover, the convolutional layers in the CNN use a set of learnable kernels or filters that each filter is applied to the specified subset of their preceding layer's output. These filters calculate feature maps in which each feature map shares the same weights. The pooling layer, also known as a subsampling layer, is a nonlinear downsampling function that condenses subsets of its input. The main goal of using pooling layers in the CNN is to reduce

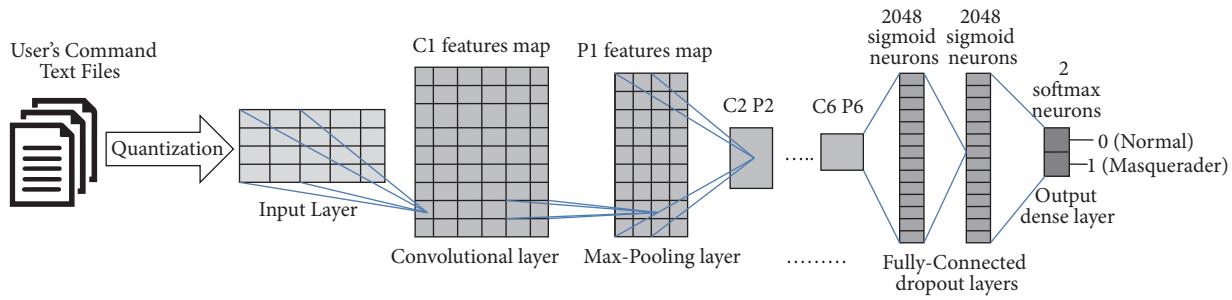

Figure 7: The architecture of the used CNN model.

the complexity and computations by reducing the size of their preceding layer's output. There are many pooling nonlinear functions that can be used, but among them, max-pooling is the mostly used which selects the maximum value in the given pooling window. Typically, each convolutional layer in the CNN is followed by a max-pooling layer. The CNN has one or more stacked convolutional layer and max-pooling layer pairs to extract features from the entire input and then map these features to their next fully connected layer. The top layers of the CNN are one or more of fully connected layers which are similar to hidden layers in the DNN. This means that neurons of the fully connected layers are connected to all neurons of the preceding layer. The output layer is the final layer in the CNN and is responsible for reporting the output value of the CNN. Finally, the back-propagation algorithm is usually used to train CNNs via Stochastic Gradient Decent (SGD) to adjust the weights of the fully connected layers [56]. There are several variant structures of CNN that are proposed in the literature, but LeNet structure which is proposed by LeCun et al. [44] is the most common approach used in many applications of computer vision and text classification.

Regarding its stability and high efficiency in text classification, we selected the CNN model which is proposed in [50] to perform a dynamic masquerade detection on all data configurations. The used model is a character-level CNN that takes a text file as input and outputs the classification score (0 if the input text file is related to a normal user or 1 otherwise). The used CNN model is from LeNet family and consists of an input layer, followed by six convolution and max-pooling pairs, followed by two fully connected layers, and finally followed by an output layer. In the input layer, the text quantization process takes place when the used model encodes all letters in the input text file using a one-hot representation from a 70-character alphabet. All the convolutional layers in the used CNN model have a ReLU nonlinear activation function. The two fully connected layers in the used CNN model are of the type dropout layer with dropout probability equal to 0.5. In addition to that, the two fully connected layers in the used CNN model have a Sigmoid nonlinear activation function as well as they have the same size of 2048 neurons of each. The output layer in the used CNN model is of the type dense layer as well as it has a softmax activation function and size of two neurons. The used CNN model is trained by back-propagation algorithm via SGD. Finally, we set the following parameters to the used CNN model: learning rate=0.01, epochs=30, and batch size=64. These values are obtained experimentally by performing a grid search to find the best possible values of these parameters. Figure 7 shows the architecture of the used CNN model and is reproduced from Zhang et al. (2015) [under the Creative Commons Attribution License/public domain].

In our work, we used a CNN model to perform a dynamic masquerade detection task on all data configurations. As mentioned in Section 5.1.1, there are six data configurations and each of them will be used in the separate experiment. So, we will have six separate CNN-experiments, and each experiment will be on one of the data configurations. The methodology of all of these experiments is the same and as follows: for the given data configuration D, we firstly prepared all the given data configuration's text files such that each file of them represents the training and test sets of a user in D. Next to that, for each user U_i in D, where i=1,2,. . ., M and M is the number of users in D, we did the following steps: we split the data of U_i into two independent sets T_i and Z_i, which are the training and test sets of the ith user in D, respectively. The splitting process followed the structure of the particular data configuration which is described in Section 3. Furthermore, we also moved each block in the training and test sets of the user U_i to a separate text file. This means that each of the training and test sets of the user U_i consists of a specified number of text files in which each text file contains one block of UNIX commands. After that, we constructed the used CNN model. The constructed CNN model is trained on T_i and then tested on Z_i. After the test process finished, we extracted and saved the outcomes: TP_i, FP_i, TN_i, and FN_i of the ith user in D. Then, we proceed to the next user in D to do the same previous steps until the last user in D is reached. After all users in D are completed, we computed the overall outcomes TP, FP, TN, and FN of the data configuration D by using (3), (4), (5), and (6), respectively. Figure 8 depicts the flowchart of the methodology of CNN-experiments.

6. Results and Discussion

We carried out three major empirical experiments, which are DNN-experiments, LSTM-RNN-experiments, and CNN-experiments. Each of them consists of six separate subexperiments where each subexperiment is performed on one of the data configurations: SEA, SEA 1v49, Greenberg Truncated, Greenberg Enriched, PU Truncated, and PU Enriched.

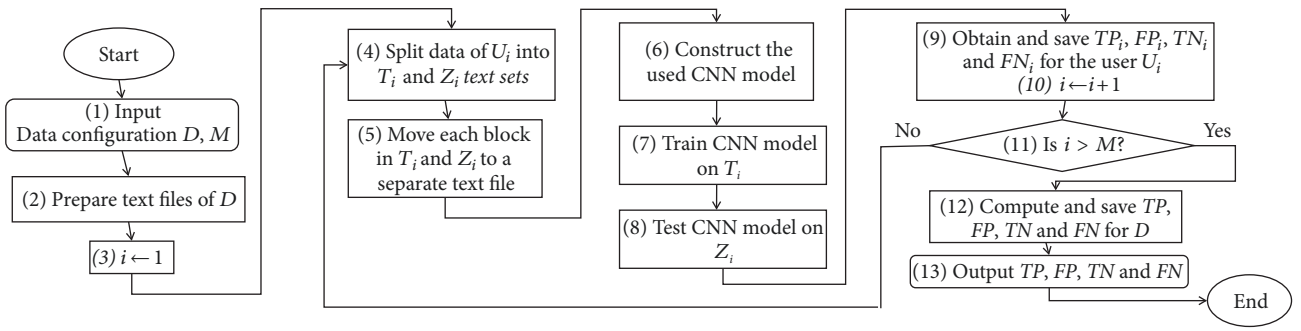

Figure 8: The flowchart of the CNN-experiments.

Table 6: The confusion matrix of the masquerade detection outcomes.

Actual Class	Predicted Class	
	Normal User	Masquerader
Normal User	TN	FP
Masquerader	FN	TP

Basically, our PSO-based DNN hyperparameters selection algorithm was implemented in Python 3.6.4 [57] with NumPy [58]. Moreover, all models (DNN, LSTM-RNN, CNN) were constructed and trained and tested based on Keras [59, 60] with TensorFlow 1.6 [61, 62] that backend over CUDA 9.0 [63] and cuDNN 7.0 [64]. In addition to that, all experiments were performed on a workstation with an Intel Core i7 CPU (3.8 GHz, 16 MB Cache), 16 GB of RAM, and the Windows 10 operating system. In order to accelerate the computations in all experiments, we also used a GPU-accelerated computing with NVIDIA Tesla K20 GPU 5 GB GDDR5. The experimental environment is processed in 64-bit mode.

In any classification task, we have four possible outcomes: True Positive (*TP*), True Negative (*TN*), False Positive (*FP*), and False Negative (*FN*). We get a *TP* when a masquerader is correctly classified as a masquerader. Whenever a good user is correctly classified as a good user itself, we say it is a *TN*. A *FP* occurs when a good user is misclassified as a masquerader. In contrast, *FN* occurs when a masquerader is misclassified as a good user. Table 6 shows the Confusion Matrix of the masquerade detection outcomes. For each data configuration, we used the obtained outcomes for that data configuration to compute twelve well-known evaluation metrics. After that by using these evaluation metrics, we assessed the performance of each deep learning model on that data configuration.

For simplicity, we divided these evaluation metrics into two categories: General Classification Measures and Masquerade Detection Measures. The General Classification Measures are metrics that are used for any classification task, namely, Accuracy, Precision, Recall, and F1-Score. On the other hand, Masquerade Detection Measures are metrics that usually are used for a masquerade or intrusion detection task, which are Hit Rate, Miss Rate, False Alarm Rate, Cost, Bayesian Detection Rate, Bayesian True Negative Rate, Geometric Mean, and Matthews Correlation Coefficient. The used evaluation metrics definition and their corresponding equations are as follows:

(i) Accuracy shows the rate of true detection over all test sets.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{7}$$

(ii) Precision shows the rate of correctly classified masqueraders from all blocks in the test set that are classified as masqueraders.

$$Precision = \frac{TP}{TP + FP} \tag{8}$$

(iii) Recall shows the rate of correctly classified masqueraders over all masquerader blocks in the test set.

$$Recall = \frac{TP}{TP + FN} \tag{9}$$

(iv) F1-Score gives information about the accuracy of a classifier regarding both Precision (P) and Recall (R) metrics.

$$F1\ Score = \frac{2}{1/P + 1/R} \tag{10}$$

(v) Hit Rate shows the rate of correctly classified masquerader blocks over all masquerader blocks presented in the test set. It is also called Hits, True Positive Rate, or Detection Rate.

$$Hit\ Rate = \frac{TP}{TP + FN} \tag{11}$$

(vi) Miss Rate is the complement of Hit Rate (Miss=100-Hit); i.e., it shows the rate of masquerade blocks that are misclassified as a normal user from all masquerade blocks in the test set. It is also called Misses or False Negative Rate.

$$Miss\ Rate = \frac{FN}{FN + TP} \tag{12}$$

(vii) False Alarm Rate (FAR) gives information about the rate of normal user blocks that are misclassified as a masquerader over all normal user blocks presented in the test set. It is also called False Positive Rate.

$$False\ Alarm\ Rate = \frac{FP}{FP + TN} \tag{13}$$

(viii) Cost is a metric that was proposed in [9] to evaluate the efficiency of a classifier concerning both Miss Rate (MR) and False Alarm Rate (FAR) metrics.

$$Cost = MR + 6 \times FAR \tag{14}$$

(ix) Bayesian Detection Rate (BDR) is a metric based on Base-Rate Fallacy problem which is addressed by S. Axelsson in 1999 [65]. Base-Rate Fallacy is a basis of Bayesian statistics and occurs when people do not take the basic rate of incidence (Base-Rate) into their account when solving problems in probabilities. Unlike Hit Rate metric, BDR shows the rate of correctly classified masquerader blocks over all test set taking into consideration the base-rate of masqueraders. Let I and I^* denote a masquerade and a normal behavior, respectively. Moreover, let A and A^* denote the predicated masquerade and normal behavior, respectively. Then, BDR can be computed as the probability P(I | A) according to (15) [65].

$$\begin{aligned} Bayesian\ Detection\ Rate &= P(I \mid A) \\ &= \frac{P(I) \times P(A \mid I)}{P(I) \times P(A \mid I) + P(I^*) \times P(A \mid I^*)} \end{aligned} \tag{15}$$

P(I) is the rate of the masquerader blocks in the test set, P(A | I) is the Hit Rate, P(I^*) is the rate of the normal blocks in the test set, and P(A | I^*) is the FAR.

(x) Bayesian True Negative Rate (BTNR) is also based on Base-Rate Fallacy and shows the rate of truly classified normal blocks over all test set in which the predicted normal behavior indicates really a normal user [65]. Let I and I^* denote a masquerade and a normal behavior, respectively. Moreover, let A and A^* denote the predicated masquerade and normal behavior, respectively. Then, BTNR can be computed as the probability P(I^* | A^*) according to (16) [65].

$$\begin{aligned} Bayesian\ True\ Negative\ Rate &= P(I^* \mid A^*) \\ &= \frac{P(I^*) \times P(A^* \mid I^*)}{P(I^*) \times P(A^* \mid I^*) + P(I) \times P(A^* \mid I)} \end{aligned} \tag{16}$$

P(I^*) is the rate of the normal blocks in the test set, P(A^* | I^*) is the True Negative Rate which is easily obtained by calculating (1-FAR), P(I) is the rate of the masquerader blocks in the test set, and P(A^* | I) is the Miss Rate.

(xi) Geometric Mean (g-mean) is a performance metric that combines true negative rate and true positive rate at one specific threshold where both the errors are considered equal. This metric has been used by several researchers for evaluating classifiers on imbalance dataset [66]. It can be computed according to (17) [67].

$$g\_mean = \sqrt{\frac{TP \times TN}{(TP + FN) \times (TN + FP)}} \tag{17}$$

(xii) Matthews Correlation Coefficient (MCC) is a performance metric that takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used even if the classes are of very different sizes (imbalance dataset) [68]. MCC has a range of −1 to 1, where −1 indicates a completely wrong binary classifier while 1 indicates a completely correct binary classifier. Unlike the other metrics discussed above, MCC takes all the cells of the Confusion Matrix into consideration in its formula which can be computed according to (18) [69].

$$\begin{aligned} &MCC \\ &= \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP + FN) \times (TP + FP) \times (TN + FP) \times (TN + FN)}} \end{aligned} \tag{18}$$

In the following two subsections, we will present our experimental results and explain them using two kinds of analyses: performance analysis and ROC curves analysis.

6.1. Performance Analysis. The effectiveness of any model to detect masqueraders depends on its values of evaluation metrics. The higher values of Accuracy, Precision, Recall, F1-Score, Hit Rate, Bayesian Detection Rate, Bayesian True Negative Rate, Geometric Mean, and Matthews Correlation Coefficient as well as the lower values of Miss Rate, False Alarm Rate, and Cost indicate an efficient classifier. The ideal classifier has Accuracy and Hit Rate values that reach 1, as well as Miss Rate and False Alarm Rate values that reach 0. Table 7 presents the percentages of the used evaluation metrics for DNN-experiments, LSTM-RNN-experiments, and CNN-experiments. Actually, the rows labeled by DNN and LSTM-RNN in Table 7 show results of the static masquerade detection by using DNN and LSTM-RNN models, respectively, whereas the rows labeled by CNN in Table 7 show results of the dynamic masquerade detection by using CNN model. Furthermore, the bold rows represent the best results among the same data configuration, whereas the underlined values are the best for all data configurations.

First of all, the impact of using our PSO-based algorithm can be seen in the obtained results of both DNN and LSTM-RNN models. The PSO-based algorithm is used to optimize the selection of DNN hyperparameters that maximized the accuracy which means that the sum of *TP* and *TN* outcomes will be increased significantly. Thus, according to (11) and (13), increasing the sum of *TP* and *TN* will lead definitely to the increase of the value of Hit as well as to the decrease of the value of FAR. Although the accuracy values of SEA 1v49 data configuration for all models are slightly lower than

Table 7: The results of our experiments.

Dataset	Data Configuration	Model	Evaluation Metrics (%)											
			Accuracy	Precision	Recall	F1-Score	Hit	Miss	FAR	Cost	BDR	BTNR	g-mean	MCC
SEA Dataset	SEA	DNN	98.08	76.26	84.85	80.33	84.85	15.15	1.28	22.83	76.25	99.26	91.52	79.45
		LSTM-RNN	98.52	82.30	86.58	84.39	86.58	13.42	0.90	18.83	82.33	99.34	92.63	83.64
		CNN	**98.84**	**87.77**	**87.01**	**87.39**	**87.01**	**12.99**	**0.59**	**16.51**	**87.72**	**99.37**	**93**	**86.78**
	SEA 1v49	DNN	96.54	99.98	96.43	98.17	96.43	3.57	0.48	6.47	99.98	52.04	97.96	70.64
		LSTM-RNN	97.86	99.98	97.79	98.87	97.79	2.21	0.38	4.48	99.98	63.70	98.7	78.74
		CNN	**98.78**	**99.99**	**98.74**	**99.36**	**98.74**	**1.26**	**0.19**	**2.40**	**99.99**	**75.51**	**99.27**	**86.22**
Greenberg Dataset	Greenberg Truncated	DNN	93.97	92.23	80.67	86.06	80.67	19.33	2.04	31.57	92.22	94.41	88.89	82.53
		LSTM-RNN	94.72	94.88	81.53	87.70	81.53	18.47	1.32	26.39	94.87	94.68	89.7	84.76
		CNN	**95.43**	**96.16**	**83.53**	**89.40**	**83.53**	**16.47**	**1.0**	**22.47**	**96.16**	**95.24**	**90.94**	**86.86**
	Greenberg Enriched	DNN	97.57	96.92	92.40	94.61	92.40	7.60	0.88	12.88	96.92	97.75	95.7	93.08
		LSTM-RNN	97.98	97.57	93.60	95.54	93.60	6.40	0.70	10.60	97.56	98.10	96.41	94.28
		CNN	**98.60**	**98.55**	**95.33**	**96.92**	**95.33**	**4.67**	**0.42**	**7.19**	**98.55**	**98.61**	**97.43**	**96.03**
PU Dataset	PU Truncated	DNN	81.0	99.59	78.61	87.86	78.61	21.39	2.25	34.89	99.59	39.49	87.66	54.63
		LSTM-RNN	82.19	99.69	79.89	88.70	79.89	20.11	1.75	30.61	99.68	41.10	88.6	56.46
		CNN	**83.75**	**99.74**	**81.64**	**89.79**	**81.64**	**18.36**	**1.50**	**27.36**	**99.73**	**43.38**	**89.68**	**58.79**
	PU Enriched	DNN	90.44	99.84	89.21	94.23	89.21	10.79	1.0	16.79	99.84	56.72	93.98	70.64
		LSTM-RNN	91.31	99.88	90.18	94.78	90.18	9.82	0.75	14.32	99.88	59.08	94.61	72.61
		CNN	**93.75**	**99.92**	**92.93**	**96.30**	**92.93**	**7.07**	**0.50**	**10.07**	**99.92**	**66.78**	**96.16**	**78.52**

the corresponding values of SEA data configuration, also Hit values are dramatically increased in SEA 1v49 for all models by 10-14% from those that are in the SEA data configuration. This is due to the structure of SEA 1v49 data configuration where there are 122500 masquerader blocks in the test set of SEA 1v49 comparing to only 231 blocks in the SEA data configuration. Moreover, the FAR values of SEA 1v49 for all models are significantly lower than the corresponding values of SEA data configuration. Hence, regarding SEA dataset, SEA 1v49 is better to use in masquerade detection than SEA data configuration.

On the other hand, as we expected, Greenberg Enriched enhanced noticeably the performance of all models in terms of all used evaluation metrics from the corresponding values of Greenberg Truncated data configuration. This can be explained by the fact that Greenberg Enriched data configuration has more information about user behavior including command name, parameters, aliases, and flags comparing to only command name in Greenberg Truncated. Therefore, regarding Greenberg dataset, Greenberg Enriched data configuration is better to use in masquerade detection than Greenberg Truncated. The same thing happened in PU dataset where its PU Enriched data configuration has better results regarding all models than PU Truncated. Thus, regarding PU dataset, PU Enriched is better to use in masquerade detection than PU Truncated data configuration.

Actually, PU Truncated and Greenberg Truncated data configurations simulate SEA and SEA 1v49 data configurations where only command name is considered. Despite that, regarding all used models, SEA 1v49 recorded the best results among the other truncated data configurations. On the other hand, PU Enriched and Greenberg Enriched are considered as enriched data configurations where extra information about users is taken into consideration. Due to that, enriched data configurations help models to build user's behavior profile more accurately than with truncated data configurations. Regarding all models, the results associated with Greenberg Enriched especially in terms of Accuracy, Hit, and FAR values are better than of the corresponding values of PU Enriched data configuration because PU dataset is very small masquerade detection dataset with a relatively low number of users (only 8 users). Also, this reason can explain why a few previous works used PU dataset in masquerade detection. However, data configurations can be sort for all used models from the upper to lower according to the obtained results as follows: SEA 1v49, Greenberg Enriched, PU Enriched, SEA, Greenberg Truncated, and PU Truncated.

For the sake of brevity and space limitation, we selected a subset of the used performance metrics in Table 7 to be shown visually in Figures 9 and 10. Figures 9(a), 9(b), 9(c), 9(d), 9(e), 9(f), 9(g), and 9(h) show Accuracy, Hit, Miss, FAR, Cost, BDR, F1-Score, and MCC percentages of the used models in each data configuration, respectively. Figures 10(a), 10(b), 10(c), 10(d), 10(e), and 10(f) show Accuracy, Hit, FAR, BDR, F1-Score, and MCC percentages for the average performance of the used models on datasets, respectively. Figures 9 and 10 can give us a visual comparison of the performance of the used deep learning models for each data configuration and dataset as well as in all datasets.

By taking an inspective look to Figures 9 and 10, we can notice the stability of deep learning models in such a way that they are enhancing masquerade detection from a data configuration to another in a consistent pattern. To explain that, we will discuss the obtained results from the perspective

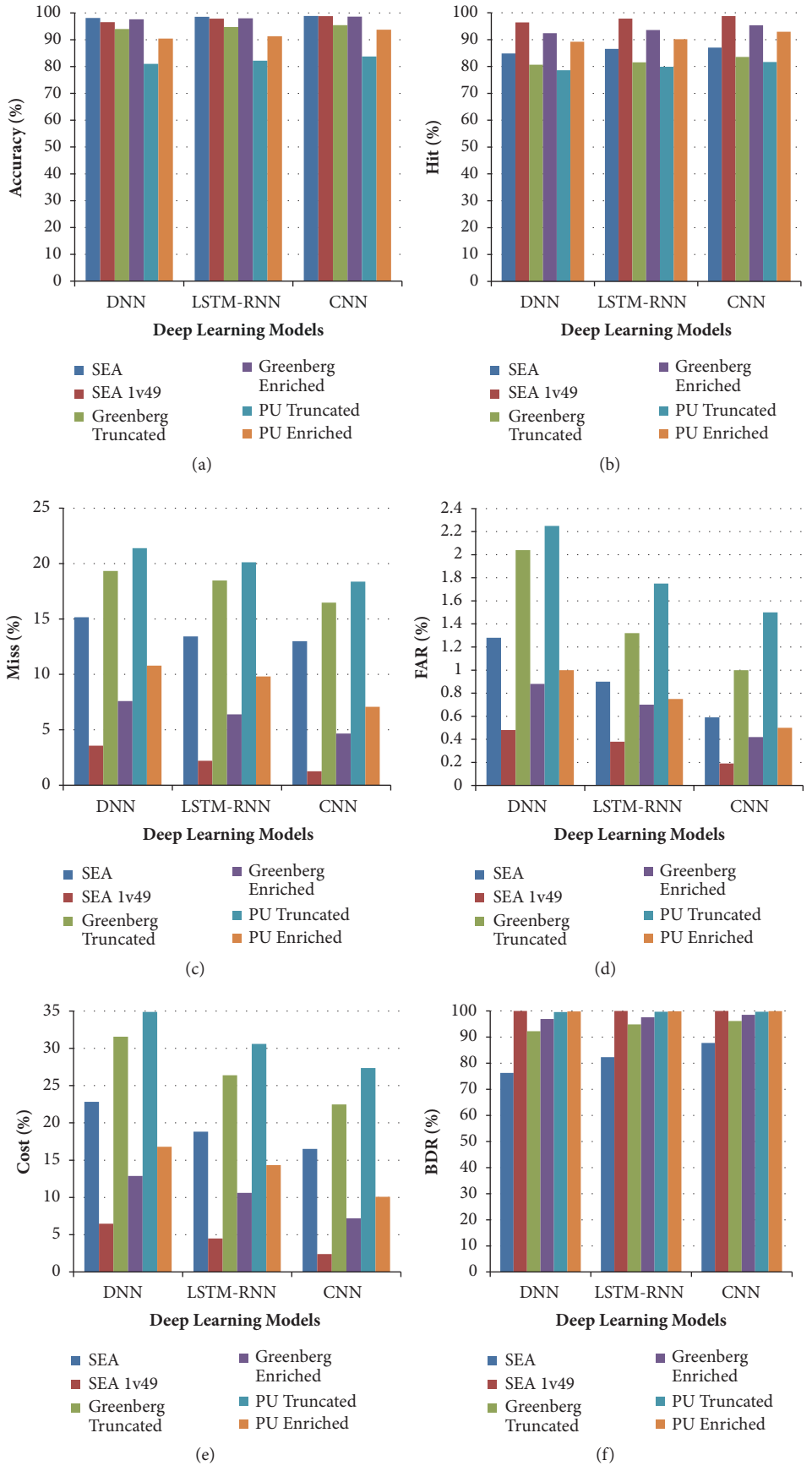

Figure 9: Continued.

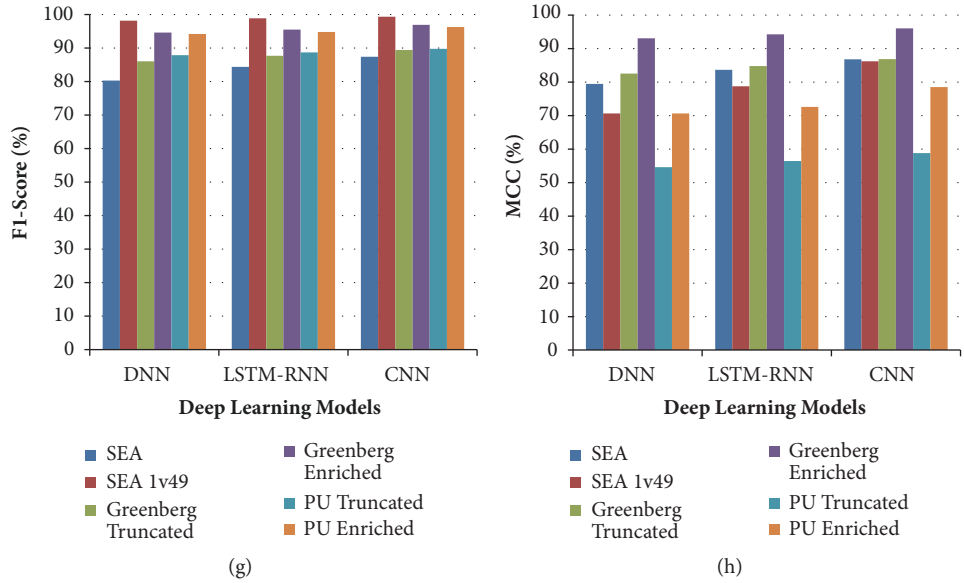

Figure 9: Evaluation metrics comparison between models on data configurations. (a) Accuracy. (b) Hit Rate. (c) Miss Rate. (d) False Alarm Rate. (e) Cost. (f) Bayesian Detection Rate. (g) F1-Score. (h) Matthews Correlation Coefficient.

of static and dynamic masquerade detection techniques. We used DNN and LSTM-RNN models to perform a static masquerade detection task on data configurations with static numeric features. The DNN, as well as LSTM-RNN, is supported with a PSO-based algorithm that optimized their hyperparameters to maximize accuracy on the given training and test sets of a user. Giving the importance to the former fact, our DNN and LSTM-RNN models output masquerade detection outcomes as better as they can reach for every user in the particular data configuration. Accordingly, at the result, their performance will be enhanced significantly on that particular data configuration. Also, this enhancement of their performance will be affected by the structure of data configuration which differs from one to another. Anyway, LSTM-RNN performed better than DNN in terms of all used evaluation metrics regarding all data configurations and datasets. This is due to the fact that LSTM-RNN model uses LSTM memory cells instead of artificial neurons in all hidden layers. Furthermore, LSTM-RNN model has self-recurrent connections as well as connections between memory cells in the same hidden layer. These characteristics of LSTM-RNN which do not exist in DNN enable LSTM-RNN to memorize the previous states, explore the dependencies between them, and finally use them along with current inputs to predict the output. However, the difference between the performance of LSTM-RNN and DNN models on all data configurations is relatively small which is between 1 and 3% for Hit and Accuracy and between 0.2 and 0.8% for FAR in all cases.

Besides static masquerade detection technique, we also used CNN model to perform a dynamic masquerade detection task on data configurations. Indeed, CNN is used in text classification task where the input is command text files for each user in the particular data configuration. The obtained results show clearly that CNN outperforms both DNN and LSTM-RNN models in terms of all used evaluation metrics on all data configurations. This is due to using a deep structure character-level CNN model which extracted and learned features from the input text files dynamically in such a way that the relation between user's individual commands can be recognized. Then, the extracted features are represented to its fully connected layers to train itself to build the user's normal profile which will be used later to detect masquerade attacks efficiently. This dynamic process and self-learning capabilities form the major objectives and strengths of such deep learning models. The used CNN model recorded very good results on all data configurations such as Accuracy between 83.75 and 98.84%, Hit between 81.64 and 98.74%, and FAR between 0.19 and 1.5%. Therefore, in our study, dynamic masquerade detection is better than static masquerade detection technique. This gives the impression that dynamic masquerade detection technique is the best choice for masquerade detection regarding UNIX command line-based datasets due to the fact that these datasets are originally textual datasets and converting them to static numeric datasets may lose them a lot of sufficient information. Despite that, DNN and LSTM-RNN also performed very well in masquerade detection on data configurations.

Regarding BDR and BTNR metrics, all the used models got high values in most cases which means that the confidence of the predicated behaviors of these models is very high. Indeed, this depends on the structure of the examined data configuration; that is, BDR will increase as much as both the number of masquerader blocks in the test set of the examined data configuration and Hit values are larger. In contrast, BTNR will increase as much as the number of normal blocks in the test set of the examined data configuration is larger and FAR value is smaller. Although all the used data configurations are imbalanced, all the used

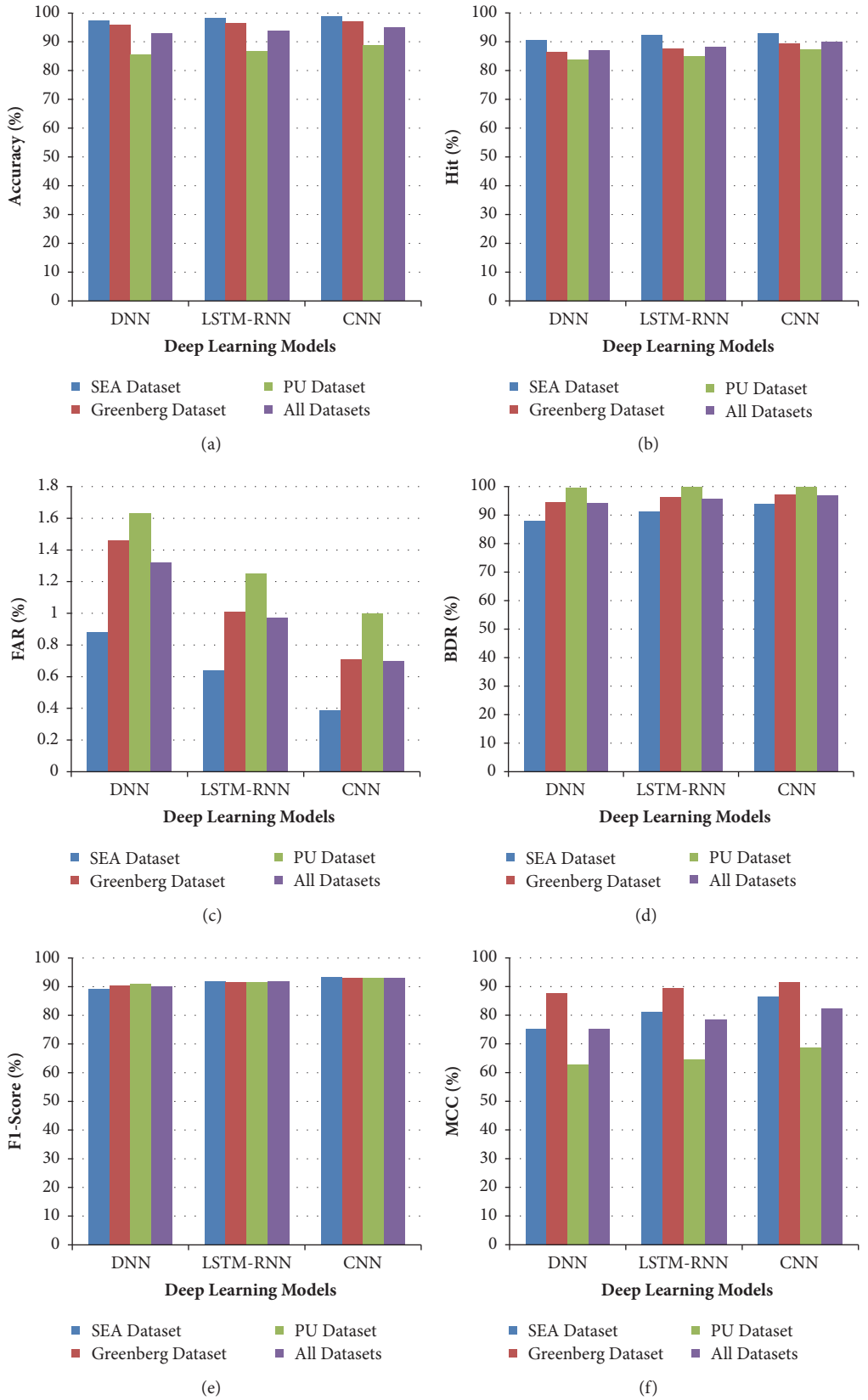

Figure 10: Evaluation metrics comparison for the average performance of the models on datasets. (a) Accuracy. (b) Hit Rate. (c) False Alarm Rate. (d) Bayesian Detection Rate. (e) F1-Score. (f) Matthews Correlation Coefficient.

Table 8: The results of statistical tests.

Measurements	Friedman Test		Wilcoxon Test					
			p1		p2		p3	
	FS	FC	W	P-value	W	P-value	W	P-value
TP	12	7	0	0.0025	0	0.0025	0	0.0025
FP	12	7	0	0.0025	0	0.0025	0	0.0025
TN	12	7	0	0.0025	0	0.0025	0	0.0025
FN	12	7	0	0.0025	0	0.0025	0	0.0025

deep learning models got high g-mean percentages for all data configurations. The same thing happened with MCC metric where all the used deep learning models recorded high percentages for all data configurations except PU Truncated.

In order to give a further inspection of the results in Table 7, we also performed two well-known statistical tests, namely, Friedman and Wilcoxon tests. The Friedman test is a nonparametric test for finding the differences between three or more repeated samples (or treatments) [70]. Nonparametric test means that the test does not assume your data comes from a particular distribution. In our case, we have three repeated treatments (k=3) each for one of the used deep learning models and six subjects (N=6) in every treatment that each subject of them is related to one of the used data configurations. The null hypothesis of Friedman test is that the treatments all have identical effects. Mathematically, we can reject the null hypothesis if and only if the calculated Friedman test statistic (FS) is larger than the critical Friedman test value (FC). On the other hand, Wilcoxon test, which refers to either the Rank Sum test or the Signed Rank test, is a nonparametric test that compares two paired groups (k=2) [71]. The test essentially calculates the difference between each set of pairs and analyzes these differences. In our case, we have six subjects (N=6) in every treatment and three paired groups, namely, p1=(DNN,LSTM-RNN), p2=(DNN,CNN), and p3=(LSTM-RNN,CNN). The null hypothesis of Wilcoxon test is the median difference of zero. Mathematically, we can reject the null hypothesis if and only if the probability (P value), which is computed using Wilcoxon test statistic (W), is smaller than a particular significance level (α). We selected α=0.05 because it is fairly common. Table 8 presents the results of Friedman and Wilcoxon tests for TP, FP, TN, and FN measurements.

It can be noticed from Table 8 that we can reject the null hypothesis of the Friedman test in all cases because FS>FC. This means that the scores of the used deep learning models for each measurement are different. One way to interpret the results of Friedman test visually is to plot the Critical Difference Diagram [72]. Figure 11 shows the Critical Difference Diagram of the used deep learning models. In our study, we got the Critical Difference (CD) value equal to 1.3533. Also from Table 8, we can reject the null hypothesis of the Wilcoxon test because P value is smaller than alpha level (0.0025<0.05) in all cases. Thus, we can say that we have statically significant evidence that medians of every paired group are different. Finally, the reason of the same results of all measurements is that models in order (CNN, LSTM-RNN, DNN) have higher scores in TP and TN as well as smaller scores in FP and FN on all data configurations.

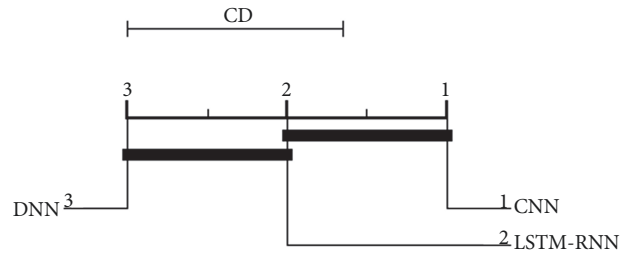

Figure 11: The Critical Difference Diagram of the used deep learning models on all data configurations.

Figures 12(a), 12(b), 12(c), 12(d), and 12(e) show comparison between the performance of traditional machine learning models and the used deep learning models in terms of Hit and FAR percentages for SEA, SEA 1v49, Greenberg Truncated, Greenberg Enriched, and PU Enriched, respectively. We obtained Hit and FAR percentages for traditional machine learning models from Table 1 as the best results in the literature. The difference between the performance of traditional machine learning and the used deep learning models can be perceived obviously. DNN, LSTM-RNN, and CNN outperformed all traditional machine learning models due to a PSO-based algorithm for hyperparameters selection used with DNN and LSTM-RNN as well as the feature learning mechanism used with CNN. In addition to that, deep learning models have deeper structures than traditional machine learning models. The used deep learning models increased considerably Hit percentages by 2-10% as well as decreased FAR percentages by 1-10% from those in traditional machine learning models in most cases.

6.2. ROC Curves Analysis. Receiver operating characteristic (ROC) curve is a plot of values of the True Positive Rate (or Hit) on Y-axis against the False Positive Rate (or FAR) on X-axis. It is widely used for evaluating the performance of different machine learning algorithms and to show the trade-off between them in order to choose the optimal classifier. The diagonal line of ROC is the reference line which means that 50% of performance is achieved. The top-left corner of ROC means the best performance with 100%. Figure 13 depicts ROC curves of the average performance of each of the used deep learning models over all data configurations. ROC

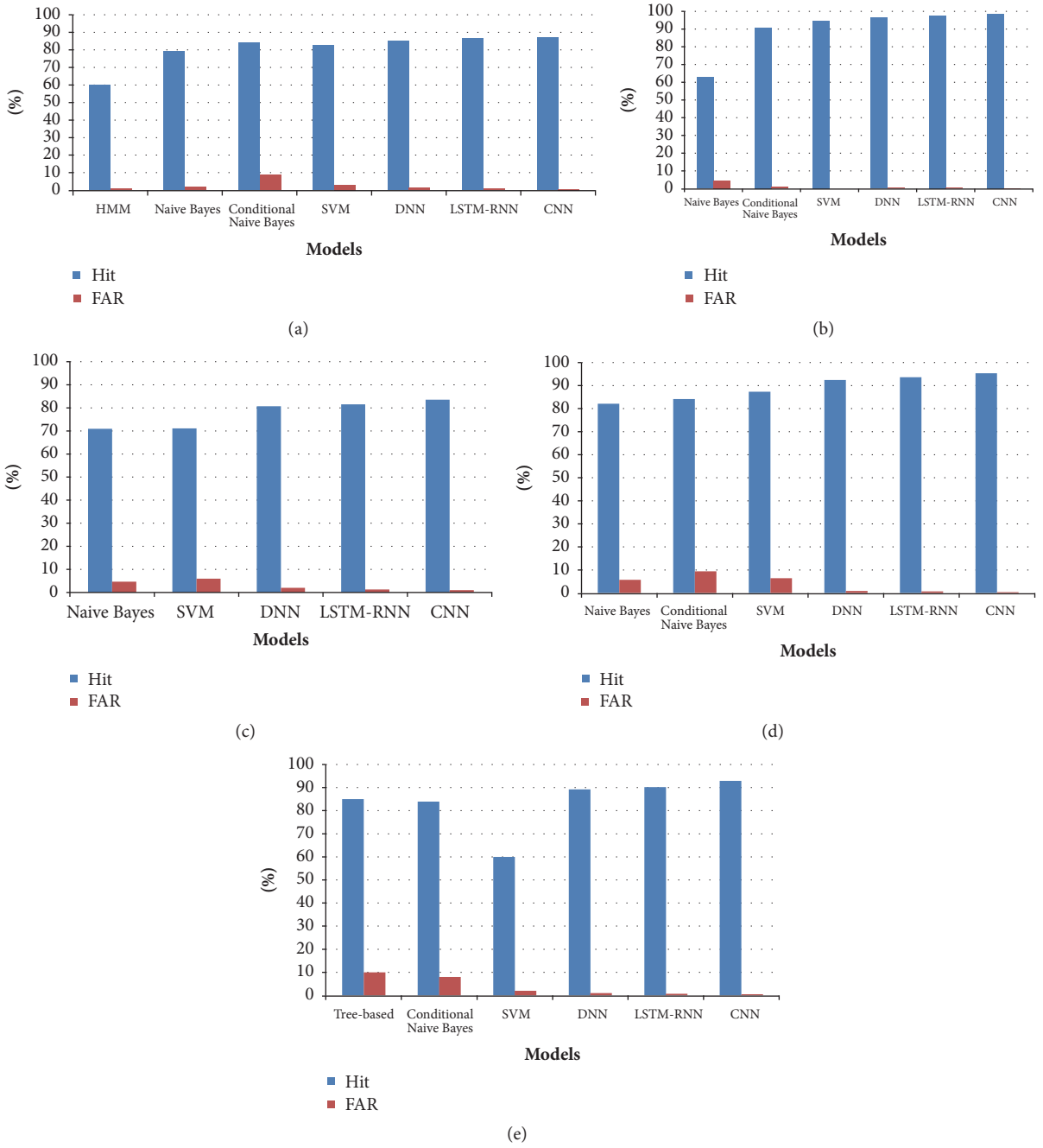

Figure 12: Models performance comparison for each data configuration. (a) SEA. (b) SEA 1v49. (c) Greenberg Truncated. (d) Greenberg Enriched. (e) PU Enriched.

curves show that models in the order CNN, LSTM-RNN, and DNN have the effective masquerade detection performance over all data configurations. However, all these three deep learning models still have a pretty good fit.

The area under curve (AUC) is also considered as a well-known measure to compare quantitatively between various ROC curves [73]. AUC value of a ROC curve should be between 0 and 1. The ideal classifier will have AUC value equal to 1. Table 9 presents AUC values of ROC curves of the used three deep learning models which are plotted in Figure 13. We can notice clearly that all these models have very high AUC values that almost reach 1, which means that their effectiveness to detect masqueraders on UNIX command line-based datasets is highly acceptable.

7. Conclusions

Masquerade detection is one of the most important issues in computer security field. Even various research studies have been focused on masquerade detection for more than one

Table 9: AUC values of ROC curves of the used models.

Model	AUC
DNN	0.9246
LSTM-RNN	0.9385
CNN	0.9617

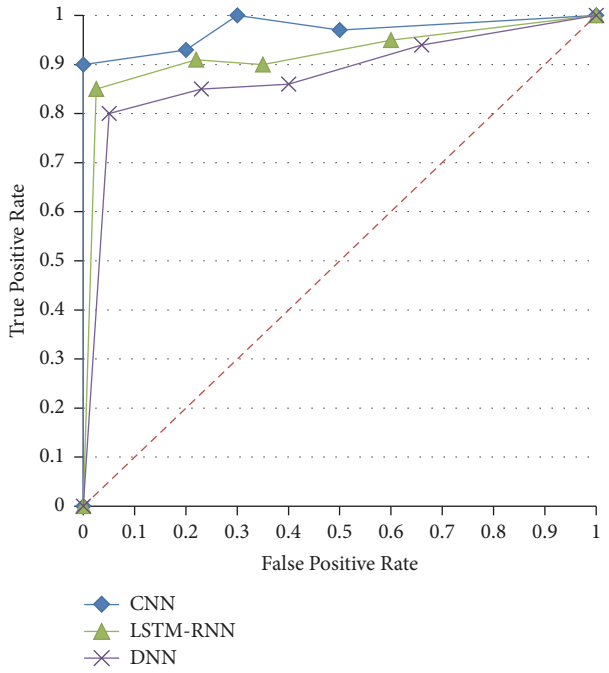

Figure 13: ROC curves of the average performance of the used models over all data configurations.

decade, but the existence of a deep study in that field utilizing deep learning models is seldom. In this paper, we presented an extensive empirical study for masquerade detection using DNN, LSTM-RNN, and CNN models. We utilized three UNIX command line datasets which are the mostly used in the literature. In addition to that, we implemented six different data configurations from these datasets. The masquerade detection on these data configurations is carried out using two approaches: the first is static and the second is dynamic. Meanwhile, the static approach is performed by using DNN and LSTM-RNN models which are applied on data configurations with static numeric features, and the dynamic approach is performed by using CNN model that extracted features from user's command text files dynamically. In order to solve the problem of hyperparameters selection as well as to gain high performance, we also proposed a PSO-based algorithm for optimizing hyperparameters of DNN. The proposed PSO-based algorithm seeks to maximize accuracy and is used in the experiments of both DNN and LSTM-RNN models. Moreover, we employed twelve well-known evaluation metrics and statistical tests to assess the performance of the used models and analyzed the experimental results using performance analysis and ROC curves analysis. Our results show that the used models performed achievement in masquerade detection regarding the used datasets and outperformed the performance of all traditional machine learning methods in terms of all evaluation metrics. Furthermore, CNN model is superior to both DNN and LSTM-RNN models on all data configurations which means that the dynamic masquerade detection is better than the static one. However, the results analyses proved the effectiveness of all used models in masquerade detection in such a way that they increased Accuracy and Hit as well as decreased FAR percentages by 1-10%. Finally, according to the results, we can argue that deep learning models seem to be highly promising tools that can be used in the cyber security field. For future work, we recommended extending this work by studying the effectiveness of deep learning models in intrusion detection for both network and cloud environments.

Data Availability

The data used to support the findings of this study are free and publicly available on Internet. UNIX command line-based datasets which are used in this study can be downloaded from the following websites: SEA dataset at http://www.schonlau.net/intrusion.html, Greenberg dataset upon a request from its owner at http://saul.cpsc.ucalgary.ca/pmwiki.php/HCIResources/UnixDataReadme, and PU dataset at http://kdd.ics.uci.edu.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

[1] L. Huang, *A study on masquerade detection*, 2010, A study on masquerade detection.

[2] M. Bertacchini and P. Fierens, "A survey on masquerader detection approaches," in *Proceedings of V Congreso Iberoamericano de Seguridad Informática*, Universidad de la República de Uruguay, 2008.

[3] R. F. Erbacher, S. Prakash, C. L. Claar, and J. Couraud, "Intrusion Detection: Detecting Masquerade Attacks Using UNIX Command Lines," in *Proceedings of the 6th Annual Security Conference*, Las Vegas, NV, USA, April 2007.

[4] L. Deng, "A tutorial survey of architectures, algorithms, and applications for deep learning," in *APSIPA Transactions on Signal and Information Processing*, vol. 3, Cambridge University Press, 2014.

[5] X. Du, Y. Cai, S. Wang, and L. Zhang, "Overview of deep learning," in *Proceedings of the 2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC)*, pp. 159–164, Wuhan, Hubei Province, China, November 2016.

[6] J. Kim, J. Kim, H. L. T. Thu, and H. Kim, "Long Short Term Memory Recurrent Neural Network Classifier for Intrusion Detection," in *Proceedings of the 3rd International Conference on Platform Technology and Service, PlatCon 2016*, Republic of Korea, February 2016.

[7] M. Schonlau, W. DuMouchel, W.-H. Ju, A. F. Karr, M. Theus, and Y. Vardi, "Computer intrusion: detecting masquerades," *Statistical Science*, vol. 16, no. 1, pp. 58–74, 2001.

[8] T. Okamoto, T. Watanabe, and Y. Ishida, "Towards an immunity-based system for detecting masqueraders," in *Proceedings of the International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, pp. 488–495, Springer, Berlin, Germany, 2003.

[9] R. A. Maxion and T. N. Townsend, "Masquerade detection using truncated command lines," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks DNS 2002*, pp. 219–228, USA, June 2002.

[10] K. Wang and S. J. Stolfo, "One-class training for masquerade detection," in *Proceedings of the Workshop on Data Mining for Computer Security*, pp. 10–19, Melbourne, FL, USA, 2003.

[11] K. H. Yung, "Using feedback to improve masquerade detection," in *Proceedings of the International Conference on Applied Cryptography and Network Security*, pp. 48–62, Springer, Berlin, Germany, 2003.

[12] K. H. Yung, "Using self-consistent naive-bayes to detect masquerades," in *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 329–340, Berlin, Germany, 2004.

[13] L. Chen and M. Aritsugi, "An svm-based masquerade detection method with online update using co-occurrence matrix," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability*, pp. 37–53, Berlin, Germany, 2006.

[14] Z. Li, L. Zhitang, and L. Bin, "Masquerade detection system based on correlation eigen matrix and support vector machine," in *Proceedings of the 2006 International Conference on Computational Intelligence and Security, ICCIAS 2006*, pp. 625–628, China, October 2006.

[15] H.-S. Kim and S.-D. Cha, "Empirical evaluation of SVM-based masquerade detection using UNIX commands," *Computers & Security*, vol. 24, no. 2, pp. 160–168, 2005.

[16] S. Greenberg, "Using Unix: Collected traces of 168 users," 88/333/45, Department of Computer Science, University of Calgary, Calgary, Canada, 1988.

[17] R. A. Maxion, "Masquerade Detection Using Enriched Command Lines," in *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, pp. 5–14, USA, June 2003.

[18] M. Yang, H. Zhang, and H. J. Cai, "Masquerade detection using string kernels," in *Proceedings of the 2007 International Conference on Wireless Communications, Networking and Mobile Computing, WiCOM 2007*, pp. 3676–3679, China, September 2007.

[19] T. Lane and C. E. Brodley, "An application of machine learning to anomaly detection," in *Proceedings of the 20th National Information Systems Security Conference*, vol. 377, pp. 366–380, Baltimore, USA, 1997.

[20] M. Gebski and R. K. Wong, "Intrusion detection via analysis and modelling of user commands," in *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery*, pp. 388–397, Berlin, Germany, 2005.

[21] K. V. Reddy and N. Pushpalatha, "Conditional naive-bayes to detect masquerades," *International Journal of Computer Science and Engineering (IJCSE)*, vol. 3, no. 3, pp. 13–22, 2014.

[22] L. Liu, J. Luo, X. Deng, and S. Li, "FPGA-based Acceleration of Deep Neural Networks Using High Level Method," in *Proceedings of the 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2015*, pp. 824–827, Poland, November 2015.

[23] J. S. Bergstra, R. Bardenet, Y. Bengio et al., "Algorithms for Hyper-Parameter optimization," *Advances in Neural Information Processing Systems*, pp. 2546–2554, 2011.

[24] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281–305, 2012.

[25] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," in *Proceedings of the 26th Annual Conference on Neural Information Processing Systems 2012, NIPS 2012*, pp. 2951–2959, USA, December 2012.

[26] O. AhmedAbdalla, A. Osman Elfaki, and Y. Mohammed AlMurtadha, "Optimizing the Multilayer Feed-Forward Artificial Neural Networks Architecture and Training Parameters using Genetic Algorithm," *International Journal of Computer Applications*, vol. 96, no. 10, pp. 42–48, 2014.

[27] S. Belharbi, R. Hérault, C. Chatelain, and S. Adam, "Deep Multi-Task Learning with evolving weights," in *Proceedings of the 24th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2016*, pp. 141–146, Belgium, April 2016.

[28] S. S. Tirumala, S. Ali, and C. P. Ramesh, "Evolving deep neural networks: A new prospect," in *Proceedings of the 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery, ICNC-FSKD 2016*, pp. 69–74, China, August 2016.

[29] O. E. David and I. Greental, "Genetic algorithms for evolving deep neural networks," in *Proceedings of the 16th Genetic and Evolutionary Computation Conference, GECCO 2014*, pp. 1451-1452, Canada, July 2014.

[30] A. Martin, F. Fuentes-Hurtado, V. Naranjo, and D. Camacho, "Evolving Deep Neural Networks architectures for Android malware classification," in *Proceedings of the 2017 IEEE Congress on Evolutionary Computation, CEC 2017*, pp. 1659–1666, Spain, June 2017.

[31] P. R. Lorenzo, J. Nalepa, M. Kawulok, L. S. Ramos, and J. R. Pastor, "Particle swarm optimization for hyper-parameter selection in deep neural networks," in *Proceedings of the 2017 Genetic and Evolutionary Computation Conference, GECCO 2017*, pp. 481–488, New York, NY, USA, July 2017.

[32] P. R. Lorenzo, J. Nalepa, L. S. Ramos, and J. R. Pastor, "Hyper-parameter selection in deep neural networks using parallel particle swarm optimization," in *Proceedings of the 2017 Genetic and Evolutionary Computation Conference Companion, GECCO 2017*, pp. 1864–1871, New York, NY, USA, July 2017.

[33] J. Nalepa and P. R. Lorenzo, "Convergence Analysis of PSO for Hyper-Parameter Selection," in *Proceedings of the International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, pp. 284–295, Springer, 2017.

[34] F. Ye and W. Du, "Particle swarm optimization-based automatic parameter selection for deep neural networks and its applications in large-scale and high-dimensional data," *PLoS ONE*, vol. 12, no. 12, p. e0188746, 2017.

[35] R. C. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Proceedings of the 6th International Symposium on Micro Machine and Human Science (MHS '95)*, pp. 39–43, Nagoya, Japan, October 1995.

[36] H. J. Escalante, M. Montes, and L. E. Sucar, "Particle swarm model selection," *Journal of Machine Learning Research*, vol. 10, pp. 405–440, 2009.

[37] Y. Shi and R. C. Eberhart, "Parameter selection in particle swarm optimization," in *Proceedings of the International conference on evolutionary programming*, pp. 591–600, Springer, Berlin, Germany, 1998.

[38] Y. Shi and R. C. Eberhart, "Emprirical study of particle swarm optimization," in *Proceedings of the 1999 congress on IEEE, Evolutionary computation, CEC 9*, vol. 3, pp. 1945–1950, 1999.

[39] J. Kennedy and R. Mendes, "Population structure and particle swarm performance," in *Proceedings of the Congress on Evolutionary Computation*, pp. 1671–1676, Honolulu, HI, USA, May 2002.

[40] M. Clerc and J. Kennedy, "The particle swarm-explosion, stability, and convergence in a multidimensional complex space," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 1, pp. 58–73, 2002.

[41] C. Yin, Y. Zhu, J. Fei, and X. He, "A Deep Learning Approach for Intrusion Detection Using Recurrent Neural Networks," *IEEE Access*, vol. 5, pp. 21954–21961, 2017.

[42] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 5, no. 2, pp. 157–166, 1994.

[43] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[44] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2323, 1998.

[45] X. Zhang and Y. LeCun, "Text Understanding from scratch," https://arxiv.org/abs/1502.01710v5.

[46] C. C. Aggarwal and C. Zhai, "A survey of text classification algorithms," in *Mining Text Data*, pp. 163–222, Springer, Boston, MA, USA, 2012.

[47] Y. Zhang and B. Wallace, "A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification," https://arxiv.org/abs/1510.03820.

[48] Y. Kim, "Convolutional neural networks for sentence classification," https://arxiv.org/abs/1408.5882.

[49] R. Johnson and T. Zhang, "Effective Use of Word Order for Text Categorization with Convolutional Neural Networks," in *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 103–112, Denver, Colorado, 2015.

[50] X. Zhang, J. Zhao, and Y. LeCun, "Character-level Convolutional Networks for Text Classification," *Advances in Neural Information Processing Systems*, pp. 649–657, 2015.

[51] K. Kowsari, D. E. Brown, M. Heidarysafa, K. Jafari Meimandi, M. S. Gerber, and L. E. Barnes, "HDLTex: Hierarchical Deep Learning for Text Classification," in *Proceedings of the 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 364–371, Cancun, Mexico, December 2017.

[52] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent Convolutional Neural Networks for Text Classification," *AAAI*, vol. 333, pp. 2267–2273, 2015.

[53] P. Liu, X. Qiu, and X. Huang, "Recurrent Neurl Network for Text Classification with Multi-Task Learning," https://arxiv.org/abs/1605.05101v1.

[54] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical attention networks for document classification," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 1480–1489, Human Language Technologies, June 2016.

[55] J. D. Prusa and T. M. Khoshgoftaar, "Improving deep neural network design with new text data representations," *Journal of Big Data*, vol. 4, no. 1, 2017.

[56] S. Albelwi and A. Mahmood, "A Framework for Designing the Architectures of Deep Convolutional Neural Networks," *Entropy*, vol. 19, no. 6, p. 242, 2017.

[57] "Python," https://www.python.org.

[58] "NumPy," http://www.numpy.org.

[59] F. Chollet, "Keras," 2015, https://github.com/fchollet/keras.

[60] "Keras," https://keras.io.

[61] M. Abadi, A. Agarwal, P. Barham et al., "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," https://arxiv.org/abs/1603.04467v2.

[62] TensorFlow, https://www.tensorflow.org.

[63] "CUDA- Compute Unified Device Architecture," https://developer.nvidia.com/about-cuda.

[64] "cuDNN- The NVIDIA CUDA Deep Neural Network library," https://developer.nvidia.com/cudnn.

[65] S. Axelsson, "Base-rate fallacy and its implications for the difficulty of intrusion detection," in *Proceedings of the 1999 6th ACM Conference on Computer and Communications Security (ACM CCS)*, pp. 1–7, November 1999.

[66] Z. Zeng and J. Gao, "Improving SVM classification with imbalance data set," in *International Conference on Neural Information Processing*, pp. 389–398, Springer, 2009.

[67] M. Kubat and S. Matwin, "Addressing the curse of imbalanced training sets: one-sided selection," in *Proceedings of the 14th International Conference on Machine Learning (ICML*, vol. 97, pp. 179–186, Nashville, USA, 1997.

[68] S. Boughorbel, F. Jarray, and M. El-Anbari, "Optimal classifier for imbalanced data using Matthews Correlation Coefficient metric," *PLoS ONE*, vol. 12, no. 6, p. e0177678, 2017.

[69] B. W. Matthews, "Comparison of the predicted and observed secondary structure of T4 phage lysozyme," *Biochimica et Biophysica Acta (BBA) - Protein Structure*, vol. 405, no. 2, pp. 442–451, 1975.

[70] W. W. Daniel, "Friedman two-way analysis of variance by ranks," in *Applied Nonparametric Statistics*, pp. 262–274, PWS-Kent Boston, 1990.

[71] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin, JSTOR*, vol. 1, no. 6, pp. 80–83, 1945.

[72] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.

[73] C. Cortes and M. Mohri, "AUC optimization vs. error rate minimization," *Advances in Neural Information Processing Systems*, pp. 313–320, 2004.