Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

A Modified Maximal Divergence Sequential Auto-Encoder And Time Delay Neural Network Models For Vulnerable Binary Codes Detection

MARWAN ALI ALBAHAR
Department of computer science - Leith, Umm Al-Qura University, Mecca, Saudi Arabia (e-mail: mabahar@uqu.edu.sa)
Corresponding author: Marwan Ali Albahar (e-mail: mabahar@uqu.edu.sa).

ABSTRACT Since the risks associated with software vulnerabilities are rapidly increasing, the detection of vulnerabilities in binary code has become an important area of concern for the software community. However, research studies associated with the detection of vulnerabilities in binary code remain limited to the handcrafted features referenced by a specific group of experts in the field. This paper considers other possibilities to add on the subject of detecting vulnerabilities in binary code. Herein, we utilize recent studies conducted on the topic of deep learning and specifically study a maximal divergence sequential auto-encoder (MDSAE) model to propose a modified version (MDSAE-NR). We also propose an altered interpretation of time-delay neural network (TDNN-NR) by incorporating a new regularization technique that produced optimized results. Finally, both models achieved good predictive performance using different evaluation metrics such as accuracy, recall, precision and F1 score compared to the baseline results. Based on the results of our experiments, we observed a 2 to 2.5% average improvement in each performance measure of interest.

INDEX TERMS Binary code vulnerability detection, Time delay neural network , Deep Learning , NDSS18 binary dataset , New regularization technique

I. INTRODUCTION

Software becomes vulnerable if it contains flaws that could create a backdoor in the software from which a hacker can gain access to a system to conduct malicious activities. These activities may include revealing or modifying critical information (e.g., Distorting, damaging the network, overwhelming the program or system) [1]. In this technological era, computer software can be found everywhere, and, due to the variety of development processes, a great deal of computers software have vulnerabilities. As a result, the issue of identifying software vulnerabilities has become a global concern. Although considerable efforts have been made by software security experts, the risk of software vulnerability has only increased over time. Numerous incidents have been identified over the past two decades, where companies and individuals were affected by harmful attacks due to software susceptibility [2]. One example of such damage is the vulnerability of a browser plugin that put the privacy of internet users at risk (e.g., Oracle Java (US-CERT 2013) and Adobe Flash Player (US-CERT 2015; Adobe Security Bulletin 2015)). Similarly, various companies and customers have been at risk of exposure and privacy breaches due to vulnerabilities found in open-source software such as Shellshock (Symantec Security Response 2014) and Heartbleed (Codenomicon 2014).

Software vulnerability detection (SVD) can be divided into two parts: vulnerability detection in binary code and source code. A wide range of studies have been conducted on the subject of vulnerability detection [3]–[8]. However, most previous studies were related to detecting vulnerabilities in source code. In particular, these studies were based on the handcrafted features selected by a limited number of experts in the domain.

Regarding the mitigation of reliance on handcrafted features, the utilization of automatic features in SVD has recently been studied [8], [9], [11]. Specifically, studies conducted in [9], [11], in which the authors employed a recurrent neural network (RNN) transformed code sequences into vector features. This was then further fed into isolated classifiers. Moreover, the authors in [8] integrated vector representation learning and classifier training in a deep network. Also , the authors in [10] studied different factors that impact the effectiveness of deep learning-based approaches for vulnerability detection. However, this study covered only the detection of vulnerabilities related to API function calls/library.

Nevertheless, vulnerability detection in binary code is more difficult than vulnerability detection in source code, since a lot of information relevant to syntax and semantics gained from high-level programming languages gets lost when the program compiles. With the help of information relevant to syntax and semantics, how data and inputs handle the execution process can be easily deduced. Unfortunately, upon analyzing code, the only item available is a binary file that contains a binary code (without the permission to access the source code) or embedded systems code. Hence, the most important feature required by the security community is the ability to detect vulnerabilities in a system from simple binary code without accessing the source code. Some previous studies have proposed methods for detecting vulnerabilities at the binary code level when the access to source code is not granted. In this context, such studies were based on symbolic execution, fuzzing [12]–[14], techniques that utilize handcrafted features derived from dynamic analysis [15]–[17], or functions similarity which helps in identifying known bugs in binaries [18].

On top of that, there were additional attempts to identify bugs in binary code. Most of the methods are either relied on the usage of semantic similarity [19] [20], supported a single architecture [21], or dynamic analysis [22], Although, these methods yielded good results, but they are inefficient due to the code coverage and expensive computation. Therefore, the attention shifted to automatic feature extraction for binary code vulnerability detection.

To the best of our knowledge, [23] is the only work has studied the use of automatically extracted features for binary code vulnerability detection. While, different studies have used automatic features combined with deep learning methodologies to detect malware [26], [27]. Despite this development, vulnerability detection in binary code remains a completely different task to malware detection. More specifically, the purpose of binary code vulnerability detection is to detect oversights and flaws in the binary code, whereas, malware detection checks if the binary code is harmful or not. The detection of vulnerabilities in binary code is more complicated than malware detection due to the slight differences between the vulnerable and non-vulnerable binaries.

Additionally, the lack of sufficient dataset has restricted research in binary code vulnerability detection. In order to solve this problem, the authors of the recent research in [23] has created a large binary dataset and made it publicly available with the aim of finding vulnerabilities in binary code. In addition, the same study has laid the foundation for this research, and we leverage recent advances in deep learning to derive a discriminant classifier that can classify both vulnerable and non-vulnerable binary functions with the highest possible accuracy.

Regularization is a key component in the machine learning field. The regularization is used during the training to reduce the error by fitting a function appropriately so that it avoids overfitting. The regularization techniques used regularly are L1 and L2. The L1 regularization puts absolute values of the magnitude of weights (coefficients) to the loss function. While the L2 regularization adds a squared magnitude of weights to the loss function [24]. Our object in this work is to present a new form of regularization technique which will add the standard deviation of the weights to the loss function instead of absolute values and squared magnitude values. The new regularization technique is based on taking the standard deviation of the weight matrix and multiplying that by λ to get the regularization term.

The main contributions of our work as follows:

- We study the maximal divergence sequential auto-encoder (MDSAE) model and propose a modified version of the MDSAE model that leverages a variational auto-encoder (VAE) and a new regularization technique for binary code vulnerability detection. The motivation behind using the MDSAE model is to learn the unknown prior of binary code alongside the classification of vulnerable and non-vulnerable code with the highest possible precision and accuracy. Furthermore, we propose a new model based on a time-delay neural network (TDNN-NR). The logic behind using a TDNN-NR is to learn the previous code relationship to determine whether the binary code is vulnerable or non-vulnerable.
- We conducted extensive experiments on the NDSS18 binary dataset. The experimental results indicate that the two variants (MDSAE-NR and TDNN-NR) outperform the baselines in all performance measures of interest.

II. BACKGROUND

A. THE VARIATIONAL AUTO-ENCODER (VAE)

Variational auto-encoder (VAE) [28] is a type of encoder that not only reconstructs true samples but also generalizes the samples generated through latent space. The main aim behind this is to train a probabilistic decoder $p_\theta(x|z), z \sim N(0, I)$ that imitates the true samples $x^1, ..., x^N$ drawn from an unknown existing distribution $p_d(x)$. The lower bound of VAE upon which it is developed is provided below.

$$\begin{aligned} \log p_\theta(x) &\geq L(x; \theta, \phi) \\ &= E_{q_\phi}(z|x)[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x) \| p(z)) \end{aligned} \tag{1}$$

$q_\phi(z|x)$ is known as the approximate posterior distribution. To maximize the log likelihood value, the objective function takes the following form for each training sample x.

$$\max_{\theta,\phi} E_x[E_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x) \| p(z)) \tag{2}$$

Where data sample x belongs to empirical data distribution. The reparametrization technique is used in order to reduce variance when applying Monte Carlo (MC) estimation for handling optimization problem given above. To be more specific, the value $q_\phi(z|x) = N(z|\mu_{\phi(x)}, diag(\sigma_{\phi(x)}))$, we can apply reparameterization as: $z = \mu_{\phi(x)} + diag(\sigma_{\phi(x)})^{1/2}\epsilon$ randomness source $\epsilon \sim N(0, I)$ and $\mu_{\phi(z)}, \sigma_{\phi(z)}$ represent the two neural networks which estimate the mean and covariance matrix of the Gaussain posterior.

The optimization function in equation 2 can be written as:

$$\begin{aligned} \max_{\phi,\theta} E_x[E_\epsilon[\log p_\theta(x|\mu_\phi(x) + diag(\sigma_\phi(x))^{1/2}\epsilon] \\ -D_{KL}(q_\phi(z|x) \| p(z))] \end{aligned} \tag{3}$$

In equation 3, it can be seen that there are two important terms. The first term is regarded as the reconstruction while the second term is regularization. Our aim is to minimize $E_x[D_{KL}(q_\phi(z|x) \| p(z))]$, for this purpose, the latent codes z need to be compressed and squashed for each true sample x such that the difference between the samples of prior distribution $p(z)$ and true samples is minimized.

B. THE KULLBACK-LEIBLER DIVERGENCE AND L2 WASSERSTEIN DISTANCE

Let we have two distribution having probability density functions of $p(z)$ and $q(z)$ where $z \in \Re^d$, the Kullback-Leibler (KL) divergence between these two distributions is given as:

$$D_{KL}(q \| p) = \int q(z) \log \frac{q(z)}{p(z)} dz \tag{4}$$

Apart from KL divergence another divergence of great importance is L2 Wasserstein (WS) distance having cost function $c(z_1, z_2) = \|z_1 - z_2\|_2^2$. The L2 WS distance between two distributions is given as:

$$D_{WS}(q \| p) = min_{\pi \in \prod(q,p)} E_{(z_1,z_2)\sim\pi}[\|z_1 - z_2\|_2^2] \tag{5}$$

Here in equation 5 the term $\prod(q, p)$ shows the set of all joint distributions over p, q which define p, q as marginal probabilities. Let p, q are the Gaussian distributions, i.e., $p(z) = N(z|\mu_1, \sum_1)$ and $q(z) = N(z|\mu_2, \sum_2)$ then both L2 WS distance and KL divergence can be calculated in close forms as:

$$\begin{aligned} D_{KL}(q \| p) = \frac{1}{2}[\log \frac{|\Sigma_1|}{|\Sigma_2|} - dtr(\Sigma_1^{-1}\Sigma_2) \\ +(\mu_1 - \mu_2)^T \Sigma_1^{-1}(\mu_1 - \mu_2)] \end{aligned} \tag{6}$$

$$D_{WS}(q \| p) = \|\mu_1 - \mu_2\|_2^2 + \|\Sigma_1^{1/2} - \Sigma_2^{1/2}\|_F^2 \tag{7}$$

where the term $\|.\|_F$ is the Frobenius norm and $\Sigma_1\Sigma_2 = \Sigma_2\Sigma_1$.

III. PROPOSED MODELS

A. THE MODIFIED MAXIMUM DIVERGENCE SEQUENTIAL AUTO-ENCODER (MDSAE) FOR BINARY VULNERABILITY DETECTION

During classification, a classifier may get over-fitted due to its high complexity. In order to reduce this classifier nature to facilitate good performance on unseen data, numerous methods are applied, such as using dropout in a neural network or using L1 and L2 regularizations. Among these methods, regularization is prominent due to its specific nature. Through regularization, we can put a penalty on loss function to get large or diverse values. In this work, we incorporated a new regularization technique based on standard deviation to the MDSAE model proposed by Le et. al [23] in order to derive a modified version. The motivation behind adding this new regularization is to add another penalty term to the existing KL divergence regularization. The first step involves making the classifier parameters more restricted through the new regularization and then studying the classification and distribution's classifier learning behaviour through new regularization. Let us assume x as the sequence of machine code, i.e. $x = [x_i]$ for $i = 1, ..., m$ where each x_i is a machine instruction. Our main aim is to make the latent codes with maximum divergence for different data classes (there are two classes in our case) by encoding x to the latent code z. Let $p_1(x)$ and $p_0(x)$ denote the distributions of vulnerable and non-vulnerable classes, respectively. We propose a modified maximum divergence auto-encoder that uses a probabilistic decoder $p_\theta(x|z)$ such as that for $z \sim p_0(z), x$ which belongs to the distribution $p_{(x|z)}$ with similar behaviour to those taken from $p_0(x)$ and for $z \sim p_1(z), x$ taken from the distribution $p_{(x|z)}$ with similar behaviour to those taken from $p_1(x)$. In other words, we are required to learn the probabilistic decoder $p_{(x|z)}$ satisfying:

$$p^0(x) = \int p_\theta(x|z)p^0(z)dz \tag{8}$$

and

$$p^1(x) = \int p_\theta(x|z)p^1(z)dz \tag{9}$$

For any approximate posterior $q_\phi(z|x)$, we have the following lower bounds: $\log p^k(x) \geq {}^k(x;\theta,\phi) = E_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x)||p^k(z)), k = 0, 1$ Further, along with KL divergence, we also embedded the new regularization. Hence, there are two regularizers in one loss function. The mathematical description of the new regularization is given below:

$$\lambda \sum_{i=1}^{k} \sigma(w_i) \tag{10}$$

where σ represent standard deviation as given below

$$\sigma(w) = \sqrt{\frac{1}{nk}\left\{\sum_{i=1}^{nk} w_i^2 - \frac{1}{nk}\left(\sum_{i=1}^{nk} w_i\right)^2\right\}} \tag{11}$$

Where λ is a regularization parameter that imposes a penalty on the regularization term to get diverse values during the training process, and n depends on the number of features in the data set. So, n is the size of the weight vector. In other words, n is the number of columns in a specific weight matrix. Similarly, k is the number of rows in the weight matrix. After embedding the new regularization, we get the following loss function L that needs to be optimized.

$$L(\theta,\phi,\omega) = -E[\log p_\phi(x|z)] + D_{KL}(q_\theta(z|x)\|p(z)) + \sigma(w) \tag{12}$$

The KL divergence keeps the representation of z and the parameters values of each data point sufficiently diverse. While the new regularization keeps the parameter values restrict in a specific std deviation to make the latent variables apart and avoid to mix them, for both vulnerable and non-vulnerable classes (See Figure 1).

To illustrate the effectiveness of the new regularization, we designed a case that included two parameters. The first parameter is presented in blue dots $W1 = parameter\ 1$ values up to 100 and the second parameter is presented in orange dots $W2 = parameter\ 2$ values up to 100. By employing the new regularization technique, it is evident that the regularizer reduces the higher values of W1 and W2. The aforementioned case used only to demonstrate the functionality of the new regularizer, and it can be considered as a preliminary result.

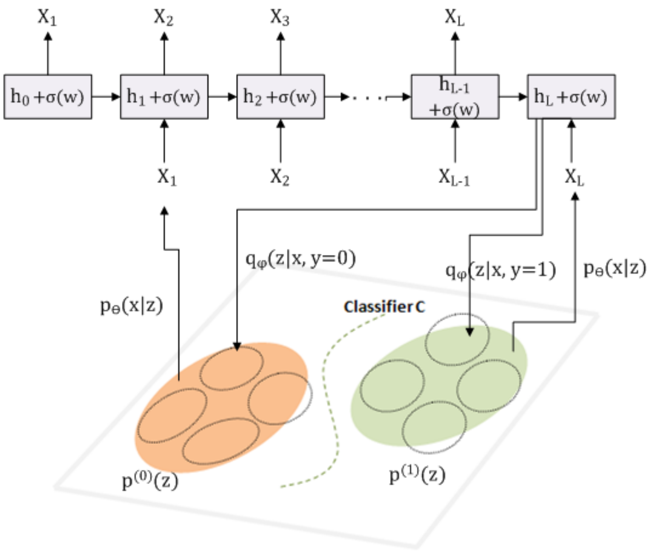


FIGURE 1: Modified Maximum Divergence Sequential Auto Encoder for binary vulnerability detection. The new regularization is integrated in hidden layer in order to make the latent code divergent and make the classifier more discriminant. The new regularization updates the weights values according to its standard deviation to avoid taking higher values during training.

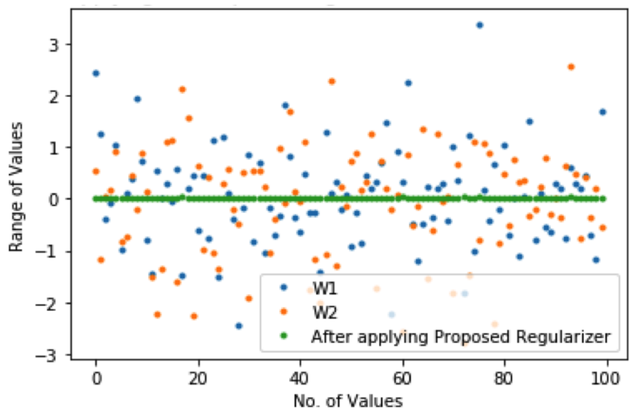


FIGURE 2: The results after applying the new regularization technique in 2D i.e. W1 and W2

B. TIME DELAY NEURAL NETWORK (TDNN) WITH NEW REGULARIZATION

Following the successful implementation of TDNN in a number of practical applications including time series prediction and speech recognition, we are interested in employing TDNN for binary SVD by identifying vulnerable and non-vulnerable classes from the binary functions. The reason for using TDNN in this work is to identify the relationship between the previous binary code (e.g. previous byte/word) in a function with the vulnerable and non-vulnerable classes. TDNN works better for time series data; therefore, it is necessary to have time series data or convert existing data to time series data. For this purpose, we assumed that, in a block of code (binaries), the previous instructions (based on the input unit of the TDNN) are considered as the previous time stamp data while the currently processing binaries are considered the current time stamp data. In other words, for each current instruction, the previous instruction acts as the previous time stamp t-1 data. As the data is fed in a sequential manner therefore, for coming instruction the prior instructions were considered as the previous.

The binary functions are not just a word/byte, but rather a sequence of words and bytes. This group of words/bytes contribute to making a function either vulnerable or non-vulnerable. In such a situation, keeping the track of prior information is essential; therefore, using TDNN is helpful because it considers the previous time steps during training. Notably, the mechanism of connections in TDNN is similar to a feed-forward neural network. The incredible difference between TDNN and neural networks is that the input to any node i in TDNN can be the output of previous nodes, not only in the current time step t but also the output of some d previous time steps, i.e. $(t-1, t-2, ..., t-d)$. This mechanism is implemented by adding tap delay lines. The activation function F in such networks for node i at time step t is given by:

$$y_i^t = F\left(\sum_{j=1}^{i-1}\sum_{k=0}^{d} y_j^{t-k} w_{ijk}\right) \tag{13}$$

Where y_i^t is the output of node i at time t, w_{ijk} is the connection strength to node i from the output of node j at time $t-k$, and F is the activation function. As these networks act like feed forward neural networks, the only difference is that the input to any hidden layer consist some previous time stamp information as well. In order to avoid complexity of the TDNN, we added the new regularization algorithm which defined in equation 11. The motivation behind adding the new regularization is to check the performance of classifier by restricting parameter values in a certain standard deviation (See Figure 3).

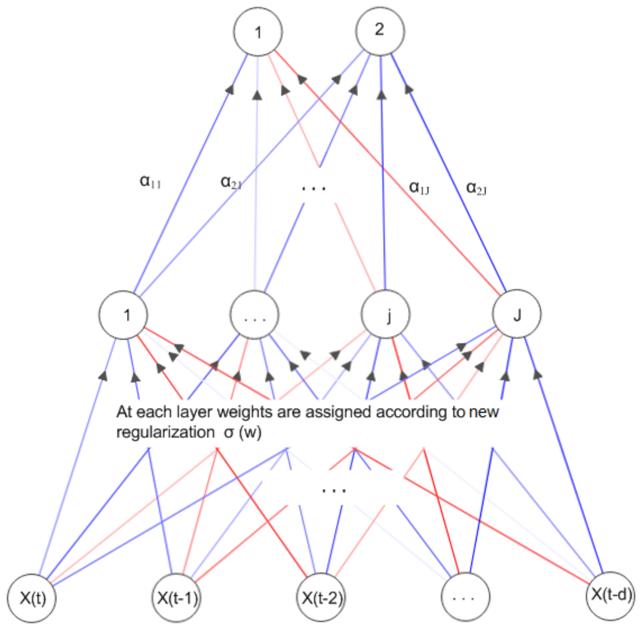

1
2
α11
α21
...
α1J
α2J
1
...
j
J
At each layer weights are assigned according to new regularization σ (w)
...
X(t)
X(t-1)
X(t-2)
...
X(t-d)

FIGURE 3: Time-delay neural network with a new regularization is trained for 100 epochs. At each layer, the new regularizer is integrated to make the TDNN more discriminant. The only difference here is that such network takes in account the previous time step values which make it more easier to find the vulnerability in the code as current byte/word is analyzed in the context of previous one.

C. PARAMETERS SETTINGS

For training the models, 80% of data is used while the remaining 20% is used for validation and testing purposes. A dynamic RNN is used to handle variations in the number of machine instructions. For both RNN baselines and the proposed models (TDNN-NR and MDSAE models), the number of hidden units was set to 512. For the proposed MDSAE model, the size of the latent space is consistent with the previous work (i.e. 4,096 units). Other parameter values are the same for both, such as α and β being set to 10^{-3} and 3×10^{-5}, respectively. An Adam optimizer was incorporated, and the learning rate was initialized to 10^{-3}. Minibatch size was set to 64 and the models were trained for 100 epochs. The proposed methods were implemented in Python using the TensorFlow framework. We performed our experiments by using Amazon Cloud Services, with an octa-core processor of frequency 2.4 GHz and Transcend 128 GB RAM.

IV. EXPERIMENT

A. DATA PROCESSING AND EMBEDDING

For each machine instruction, we followed the same process proposed in [23] with the Capstone2 binary disassembly framework being incorporated to detect entire machine instruction. Furthermore, redundant prefixes were removed to obtain the essential parts consisting of opcode and other significant information. There are mainly two parts in a machine instruction: the opcode and instruction information (i.e. operands, registers, memory addresses, etc.). Opcode and instruction information were embedded into vectors and then concatenated. To embed opcode, a vocabulary of opcodes was created and then multiplied by the one-hot vector of the opcode with a corresponding

	Vulnerable	Non-Vulnerable	Binaries
Windows	8,978	8,999	17,977
Linux	7,349	6,955	14, 304
Whole	16,327	15,954	32,281

TABLE 1: Dataset used for experimental purpose

embedding matrix. A vocabulary of over 256 hex-bytes from 00 to FF was created to embed the instruction information and were then multiplied by the corresponding embedding matrix. More precisely, the output of the process is $I = I_{op}||I_{ii}$ where I_{op} = one-hot$(op) \times M^{op}$ and $I_{ii} = freq(ii) \times M^{ii}$. Where op is the opcode, ii is the instruction information and M^{op} and M^{ii} are the corresponding embedding matrices.

B. EXPERIMENTAL DATASET

To test our model, we used the dataset extracted by Le et. al [23]. They used the NDSS18 dataset and extracted functions from this source code. Furthermore, they obtained approximately 13,000 identical functions, of which 9,000 fixed functions were converted to binary using their automatic tool. Next, they dissected the semantic and syntactical relationship using their developed tool, Joern3, in the selected source code. They compiled various vulnerable and non-vulnerable functions for different platforms and obtained a total of 32,281 binary functions, of which 17,977 binaries compiled for Windows and 14,304 compiled for the Linux platform.(see Table 1 for the detailed dataset).

C. EXPERIMENTAL RESULTS

We conducted experiments using our proposed models on a subset of Windows and Linux as well as the entire set of binaries. Then, we compared our proposed models with other state-of-the-art methods. The experimental results are shown in Table 2, 3 and 4.

To compare MDSAE-NR and TDN-NR model, it is clear that MDSAE-NR performed well in terms of accuracy and precision. Accuracy is a biased term which can be altered by tweaking the data, but on other hand precision and recall are the generally accepted measure. Precision measure detected true positives out of all positives instances. The accuracy and precision of the MDSAE-NR model are 1.2% higher than the TDN-NR model. As far as F1 score is a concern, the results for both models are almost the same as it depends upon the two measures, precision and recall. Similarly, AUC-ROC is an important measure to analyze the performance of the classification model. ROC is a curve drawn over different probabilities while AUC is the area under the ROC curve which shows the degree of separability of the model. In other words, the higher the value of AUC-ROC, the better the model is in distinguishing between different classes. In our case, the AUC-ROC is almost the same and higher than 85% which denotes that the separation ability of our models is also higher. Based on these results, it is evident that our proposed models outperform the baselines in all performance measures of interest. Specifically, in the field of computer security, recall is a very important measure of completeness since a higher recall value leads to fewer vulnerable functions being incorrectly classified as non-vulnerable, which can otherwise present an issue for code auditors due to a large imbalance in the number of non-vulnerable and vulnerable functions in real-world use. In addition, since the resulting data representations of both models work well with a linear classifier, this confirms that our proposed models efficiently support the classifiers to achieve good results. The main advantage of the MDSAE-NR is that it first learns the distribution of both vulnerable and non-vulnerable priors and makes them separable from each other with higher accuracy, as shown in Figures 7 and 8 due to the employment of VAE, which learns the distribution of data. In addition, MDSAE-NR attempts to make both of the priors consistent and gradually more distant instead of attempting to directly classify as per other existing neural networks. Similarly, the TDNN-NR model also has the advantage of keeping track of the previous time stamp data based on the previous binaries, from which it infers the results on whether binary data is vulnerable or non-vulnerable. Overall, through extensive experimentation, we observed similar changes to [23]. Particularly, the vulnerable binary code and fixed version of it differ by a very few machine instructions. Since the models are required to pay attention to these minute differences to reconstruct them, the ability to reconstruct a vulnerable binary and its fixed version in the latent space is crucial. Moreover, we also observed that our proposed models make both priors more distant than the previous work, as evident from the experimental results.

Method	Acc	Rec	Pre	F1	AUC-ROC
MDSAE-NR	86.4	98.2	79.5	89.0	86.3
TDNN-NR	85.2	97.9	78.3	87.1	85.4
RNN-R [23]	54.1	92.6	52.6	67.0	53.8
Para2Vec [25]	55.5	93.5	53.4	68.0	55.0
MD-RKL [23]	80.8	86.9	77.6	82.0	80.7
MD-RWS [23]	80.6	91.3	75.5	82.6	80.6
RNN-C [23]	81.5	94.6	75.1	83.7	81.4
VulDeePeck [8]	82.5	94.4	76.5	84.5	82.4
SeqVAE-C [23]	80.8	91.4	75.7	82.8	80.7
MD-CKL [23]	83.2	97.7	75.8	85.4	83.0
MD-CWS [23]	84.5	97.2	77.7	86.4	84.4

TABLE 2: Experimental results for Windows platform from NDSS18 binary dataset in percentage

Method	Acc	Rec	Pre	F1	AUC-ROC
MDSAE-NR	88.6	99.1	84.4	90.2	87.7
TDNN-NR	87.3	98.9	84.1	89.3	87.4
RNN-R	55.3	93.5	53.3	67.9	54.9
Para2Vec	55.8	92.1	53.6	67.8	55.5
MD-RKL	82.7	81.3	83.9	82.6	82.7
MD-RWS	84.7	90.7	81.2	85.7	84.6
RNN-C	84.4	96.9	77.7	86.3	84.2
VulDeePeck	85.5	94.2	80.5	86.8	85.4
SeqVAE-C	83.0	93.7	77.5	84.8	82.9
MD-CKL	85.9	97.2	79.5	87.4	85.7
MD-CWS	86.9	97.8	80.6	88.3	86.8

TABLE 3: Experimental results for Linux platform from NDSS18 binary dataset in percentage

Method	Acc	Rec	Pre	F1	AUC-ROC
MDSAE-NR	87.5	99.3	81.2	89.8	87.1
TDNN-NR	86.6	98.7	80.3	88.3	86.3
RNN-R	56.3	93.9	53.9	68.5	55.8
Para2Vec	54.9	94.3	53.1	67.7	54.4
MD-RKL	75.3	87.8	70.5	78.2	75.1
MD-RWS	83.7	94.3	78.0	85.4	83.5
RNN-C	83.4	94.1	77.8	85.2	83.3
VulDeePeck	83.5	91.0	79.5	84.8	83.4
SeqVAE-C	78.5	89.4	73.6	80.7	78.4
MD-CKL	82.3	98.0	74.8	84.8	82.1
MD-CWS	85.3	98.1	78.4	87.1	85.2

TABLE 4: Experimental results for whole dataset in percentage

D. INSPECTION OF MODELS BEHAVIORS

1) Distances between Two Priors, Distributions of Vulnerable, Non-vulnerable Classes During Training

During our experiment, we analyzed the following measures for MDSAE-NR model:

- Euclidean Distance of two means of priors, that is $\|\mu_0 - \mu_1\|$.
- WS Distance between two priors.
- Reconstruction Loss: As can be seen from Figure 4 a and b that the Euclidean distance and WS distance of the two prior increasing during training process which sets the two distributions apart as depicted in figure 6, 7 and 8. As both the distances between the two prior increase, the discrimination power of the classifier also increases and hence classify both the vulnerable and non-vulnerable classes with high precision and accuracy. From Figure 4 c, it is clearly visible that the reconstruction loss gradually decreasing and the latent codes retain the necessary information in binaries during training process.

For TDNN-NR, we studied the following measure:

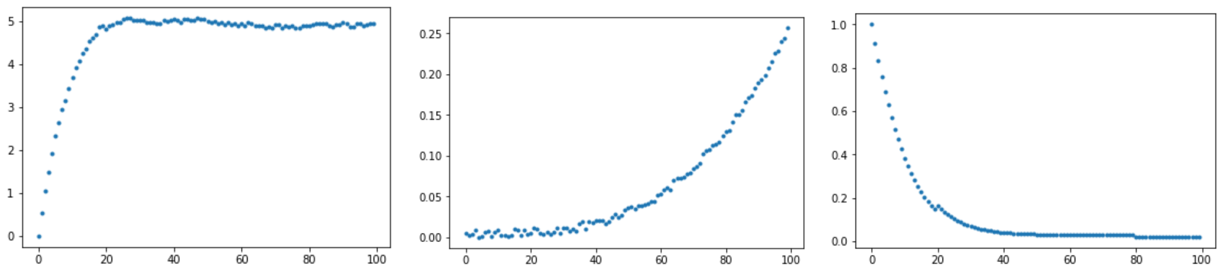
- AUC-ROC Curve: The AUC-ROC Curve result of applying TDNN-NR model for the entire dataset is shown in Figure 5. It is apparent that the performance of TDNN-NR is slightly lower than the MDSAE-NR model. The reason behind this is the absence of latent codes and time-delay neural network learns the data representation and the classification task simultaneously and tunes the parameters accordingly, which is not much efficient.

2) Visualization in the latent space of latent codes of two classes

The dimension of latent space is set to 2 in order to visualize the two classes' latent codes after and before training. Before training, the codes of both classes were mixed and very difficult to separate. Since the epochs (training process) go on to 50 and then to 100, the codes were separated and can thus be seen as separable and distinct. The visualization is shown in Figures 6, 7 and 8. This separation of codes demonstrates that the proposed model has learned the data representation and classified the two classes with high precision and accuracy. From Figure 8, it is evident that the separation of the two codes is very high, thereby leading to satisfactory results.

V. CONCLUSION

The detection of vulnerabilities in binary code is an important issue to solve in the software industry and in the field of computer security. In this paper, we proposed two different models. The first model was a modified version of Le et al.'s [23] model, while the second model was based on TDNN. The dataset used for the experiment was NDSS18, which contains different binary vulnerable and non-vulnerable functions for Windows and Linux platforms. A new regularizer is integrated into these two models in order to make both models more discriminant by making the parameter values restricted in a specific standard deviation. The results of both models were compared to the recent study by Le et. al. [23]. For the first model, on average, we achieved a 2% performance increase. For the time-delay network, we achieved an approximately 1.4% improvement in various performance measures, as seen in Table 2, 3 and 4. The advantage of both of these models is that, for a specific performance measure, one can use only a single model. Therefore, there is no need to use a different model for different higher performance measures, unlike in the work of Le et al. [23]. As such, using our trained models can provide satisfactory results for all performance measures such as precision, TPR and FPR, among others. The reason behind this good performance is that the latent code in MMDSAE-NR makes the prior maximally different and maintains crucial information in data. Similarly, the TDNN-NR keeps track of the information in previous data while leveraging the concept of restricting the parameter values in a certain standard deviation, as introduced by the new regularization.



(a) Euclidean Distance (b) WS Distance (c) Reconstruction Loss

FIGURE 4: The measures used to inspect MDSAE-NR behavior

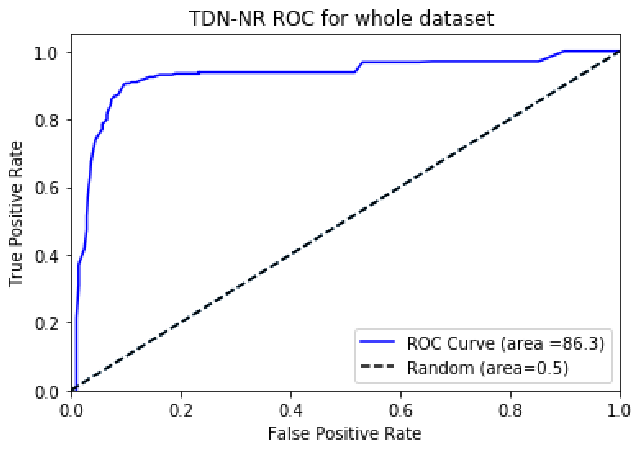


FIGURE 5: AUC-ROC Curve for whole dataset

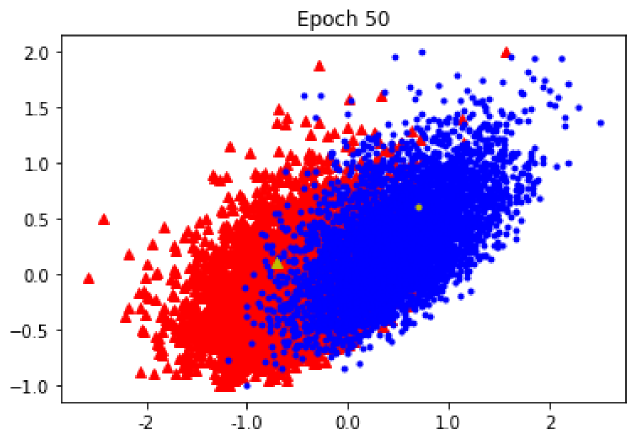


FIGURE 7: 2D latent codes during training: The yellow points are the average points of the codes

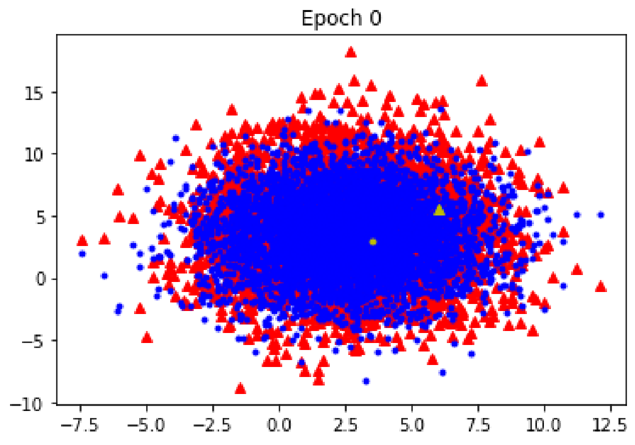


FIGURE 6: 2D latent codes before training: The yellow points are the average points of the codes

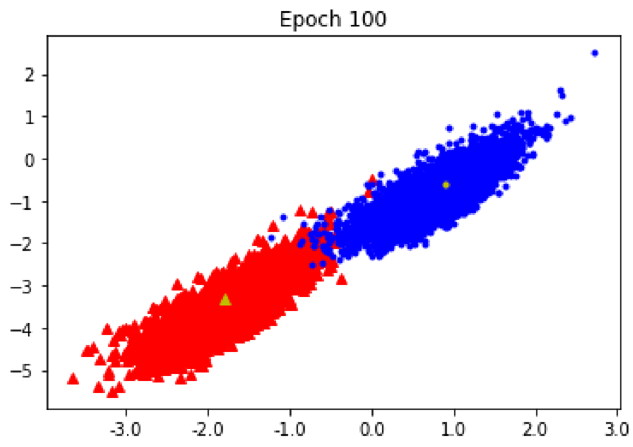


FIGURE 8: 2D latent codes after training: The yellow points are the average points of the codes

REFERENCES

[1] M. Dowd, J. McDonald, and J. Schuh. The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Addison-Wesley Professional, 2006. ISBN 0321444426.
[2] S. M. Ghaffarian and H. R. Shahriari. Software vulnerability analysis and discovery using machinelearning and data-mining techniques: A survey. ACM Computing Surveys (CSUR), 50(4):56, 2017.
[3] Y. Shin, A. Meneely, L. Williams, and J A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of

software vulnerabilities. IEEE Transactions on Software Engineering, 37(6):772–787, 2011.

[4] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07, pp. 529–540, 2007. ISBN 978-1-59593-703-2

[5] F. Yamaguchi, F. Lindner, and K. Rieck. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In Proceedings of the 5th USENIX conference on Offensive technologies, pp. 13–23, 2011.

[6] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu. VulPecker: An automated vulnerability detection system based on code similarity analysis. In Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC '16, pp. 201–213, 2016. ISBN 978-1-4503-4771-6.

[7] S. Kim, S. Woo, H. Lee, and H. Oh. VUDDY: A scalable approach for vulnerable code clone discovery. In IEEE Symposium on Security and Privacy, pp. 595–614. IEEE Computer Society, 2017.

[8] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. VulDeePecker: A deep learning-based system for vulnerability detection. CoRR, abs/1801.01681, 2018.

[9] H. K. Dam, T. Tran, T. Pham, N. S. Wee, J. Grundy, and A. Ghose. Automatic feature learning for vulnerability prediction. CoRR, abs/1708.02368, 2017.

[10] Z. Li, D. Zou, J. Tang, Z. Zhang, M. Sun, and H. Jin, A Comparative Study of Deep Learning-Based Vulnerability Detection System, IEEE Access, vol. 7, pp. 103184-103197, 2019.

[11] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague. Cross-project transfer representation learning for vulnerable function discovery. In IEEE Transactions on Industrial Informatics, 2018.

[12] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. Communications of the ACM, 56(2):82–90, 2013.

[13] A. Avancini and M. Ceccato. Comparison and integration of genetic algorithms and dynamic symbolic execution for security testing of cross-site scripting vulnerabilities. Information and Software Technology, 55(12):2209-2222, 2013.

[14] Q. Meng, S. Wen, B. Zhang, and C. Tang. Automatically discover vulnerability through similar functions. In Progress in Electromagnetic Research Symposium (PIERS), pp. 3657–3661. IEEE, 2016.

[15] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier. Toward large-scale vulnerability discovery using machine learning. In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, CODASPY '16, pp. 85–96, 2016. ISBN 978-1-4503-3935-3.

[16] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In OSDI, volume 8, pp.255–266, 2008.

[17] D. H. White and G. Luttgen. Identifying dynamic data structures by learning evolving patterns in memory. In TACAS, pp. 354–369. Springer, 2013.

[18] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code, in Proceedings 2016 Network and Distributed System Security Symposium, 2016.

[19] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross Architecture Bug Search in Binary Executables. In Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P), 2015.

[20] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu. Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2014.

[21] D. Gao, M. K. Reiter, , and D. Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In Proceedings of the 4th International Conference on Information Systems Security, 2008.

[22] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In Proceedings of the 23rd USENIX Security Symposium, 2014.

[23] T. Le and T. Nguyen and T. Le and D. Phung and P. Montague and O. De Vel and L. Qu, Maximal Divergence Sequential Autoencoder for Binary Software Vulnerability Detection,International Conference on Learning Representations,2019, https://openreview.net/forum?id=ByloIiCqYQ.

[24] P. Murugan, S. Durairaj , Regularization and Optimization Strategies in Deep Convolutional Neural Network, arXiv Preprint 2017, arXiv:1712.04711.

[25] V. Le and T. Mikolov. Distributed representations of sentences and documents. In International on Machine Learning 2014, volume 32 of JMLR Workshop and Conference Proceedings, pp. 1188–1196. JMLR.org, 2014.

[26] J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. In Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on, pp. 11-20. IEEE, 2015.

[27] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas. Malware detection by eating a whole exe. arXiv preprint arXiv:1710.09435, 2017.

[28] D. P. Kingma and M. Welling. Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114, 2013.

• • •