Reichman University Efi Arazi School of Computer Science

Reichman University
Efi Arazi School of Computer Science
M.Sc. program

Coping with Covariate Shift in Malware Classifiers

by

Nir Rosen

Final project, submitted in partial fulfillment of the requirements for the M.Sc. degree, School of Computer Science
Reichman University (The Interdisciplinary Center, Herzliya)

December 2021

This work was carried out under the supervision of Dr. Liron Allerhand from Tel-Aviv University and Dr. Ilan Gronau from the Efi Arazi School of Computer Science, Reichman University.

Abstract

Malware detection is one of the biggest challenges in the field of cyber security. Nowadays there is an extensive use of deep learning in order to detect or classify malware. However, malware changes quickly in order to avoid detection, which makes covariate shifts a major challenge in malware detection. The term covariate shift refers to changes in the distribution of the data. This poses great challenges in both training and validating machine learning classifiers in the field of malware detection, as well as limits the relevance of such classifiers in the long-term.

I present a known practice of time-based train-test split, which is often referred to as evaluation through time. I demonstrate this practice in the field of malware detection and show that it provides good performance estimates in the presence of covariate shifts.

I also present a new method that aims to train and evaluate deep learning models to make them robust to covariate shift and provide good performance estimates under covariate shift. This is achieved by changing the training method without any changes to the network architecture.

My method changes the classic training and evaluation procedure in two key ways. First of all, I refrain from shuffling the samples in the training set and instead feed them to the network sorted by time. This creates an inductive bias towards patterns that appear in later samples and helps the model to generalize to the latest patterns.

Secondly, I split the data to train, validation and test sets by time and retrain the model twice. The first retrain is used to optimize hyper parameters based on the test-set performance and the second retrain is done to train the final model on all the data, including the latest samples. To evaluate my results I kept a ”future set”, which is disjoint from all other data and contains the latest samples.

I tested my method on nine different deep learning architectures and calculated the confidence intervals of the AUC to prove statistical difference for any improvement in performance. I observed that the method preserved good performance estimates, while also improving the actual performance of three out of the nine network architectures I tried.

Contents

Chapter 1

Introduction

Malware detection is one of the biggest challenges in the field of cyber security. The term malware refers to malicious software. Malware can be detected either as a file or as a process.

Malware files are usually Portable Executable (PE) files, which are the files that are used to make some activity within the Operating System. The most known type of PE is .exe file. A process is an entity created within the OS, to perform some activity. A PE is therefore a passive recipe used by the OS to create a process, which is an active entity. The term sample refers to PEs that either are containing malware or are suspected as such. Malware sample refers to a PE which is known to contain malware.

Malware detection received considerable attention in recent years [5] [25] [21] [23] [16]. Malware detection requires large scale analysis of files across various operating systems. Malware analysis is based on either analysing a file without running it, which is referred to as static analysis, or on tracking a file when it runs on a machine, which is referred to as dynamic analysis.

Traditional malware detection relies on simple string signatures, byte-sequence signatures and repositories of known malicious files (also known as blacklists) [25]. These methods are unable to detect unknown malware [25] [19] and they require a considerable amount of expensive manual analysis by highly trained professionals to find signatures for new malware.

Behaviorial heuristics are a newer approach to malware detection. As the name suggests, behaviorial heuristics monitor the behavior of a process while it is being executed and apply some heuristic to determine whether or not it is malicious.

Modern, so called Next-Gen Anti-Virus (AV) software, uses machine learning to classify process as malicious or benign. In this work I refer to machine learning classifiers for malware detection as malware classifiers.

Malware classifiers can use dynamic and static features as inputs. As mentioned before, dynamic features are extracted from a file being executed and are mainly focused on the behavior of the process and its interaction with the operating system. In this work I will focus on dynamic malware classifiers.

A process interacts with the operating system through system Application Programming Interfaces (APIs) and system calls. System APIs are operating system functions which are located on the user-mode portion of the system. For example the ’OpenFile’ System API, which is located on the user-mode Dynamic-Link Library (DLL) file ’Ntdll’ [6]. System calls are the operating system functions which are located on the kernel-mode portion of the system. For example the 'NtOpenFile' System call, which is located on the kernel-mode executable file 'ntoskrnl' [6]. In this work I refer to both system APIs and system calls as syscalls.

Syscalls are often used for dynamic malware classification [5] [25] [23]. Many syscalls based malware classifiers have been suggested in the literature. Some uses statistical machine learning methods such as SVM, Random Forest and Naive Bayes [17] [22] [9] [26]. Following the rise of deep learning, several deep learning methods have been used for malware classification including Convolutional Neural Network (CNN) and Long Short-Term Memory (LSTM) networks [17] [22] [20].

Unlike most applications of machine learning, malware detection entails inherent covariate shifts. The term covariate shift refers to changes in the distribution of the data. In the malware detection domain, one type of covariate shift is ratio shift, in which changes in the prior manifest are reflected in changes in the ratio of malicious to benign processes. Another type of covariate shift in this domain is behaviour shift, in which changes in the distribution of the classes manifest are expressed as changes in the behavior of malware and benign processes. This poses great challenges in both training and validating machine learning classifiers in the field of malware detection, as well as limits the relevance of such classifiers in the long-term.

Despite the great progress in the field of malware classifiers, covariate-shifts are rarely taken into account. In this work I will focus on the behaviour shift that malware detectors are facing when dealing with outdated data "in the lab" versus present-day examples on production. I will study methods which can be used with deep learning models to minimize the effect of covariate shifts on the performance of a classifier.

The novelties of this work are:

1. Introduce a way to split time dependent datasets to provide a good evaluation while preserving and even improving performance.
2. Introduce a way to mitigate the effect of covariate shifts on the performance of a classifier using new training and evaluation methods.
3. Demonstrate the above in the field of malware detection.

Chapter 2

Related work

Syscalls based malware classifiers received considerable attention in recent years [17] [22] [9] [20]. The survey conducted in [5] surveyed various detection techniques applied to different data sources such as API call graph, byte sequence, PE header and sections, opcode sequence frequency and kernel features and functions. In the survey, the most common data source used was API-calls by a large margin. The authors of [5] also proposed an approach that focuses on N-grams of API calls with Naïve Bayes as their machine learning model.

Another survey [25] compared 25 studies from the years 2001-2016 which categorize file samples into different classes based on the analysis of feature representations. In their comparison, no less than half of the papers were based on API and System calls.

In the wide context of covariate shifts, the paper [14] discussed performance evaluation and model uncertainty under covariate shift and out-of-distribution data. Another paper [13] discussed covariate shift and how it effects models performance in production.

Covariate shifts can be addressed with chronological evaluation. In a situation where the samples in the data have a significant occurrence time, the known practice of a randomized train-test split brings in ”future-knowledge” into the train set. I.e., the classifier trains on samples which occurred before the samples that the classifier is evaluated with. Such evaluation is unrealistic and causes overly optimistic results. Chronological evaluation is therefore a time-based split which is not using randomization.

To the best of my knowledge, within the context of malware classifiers, chronological evaluation was first discussed in the paper [18]. In their experiment they used a dataset including files from the years 2000 to 2007, trained on files from each year until year k and tested on the following years. They showed a clear trend in which the performance improves when the training set is updated on a yearly basis and concluded that the performance of a malware classifier improves as the training set is more recent.

Another recent paper [15] showed that covariate shifts lead to biases in the performance evaluation of malware classifiers, due to the way the train-test split is done and suggested a time-based split would prevent this bias. It further claims that the evaluation of a malware classifier is also biased by the unrealistic ratio of malware to benign software used to evaluate the classifiers and suggested to define a realistic ratio in the testing set while tuning the ratio in the training set.

Another paper [8] pointed out that the fast evolution of malware makes malware classifiers unsustainable in the long run without retraining. It proposed a framework to identify aging classifiers before their performance deteriorates considerably. Based on that, the paper [4] proposed a rejection framework in which examples that are likely to be misclassified are instead quarantined until they can be expertly analyzed.

The paper [11] claims that using labels which were found many months after samples were first detected, essentially involves future knowledge and therefore inflates measured detection. It therefore argues that using cross validation disregards temporal consistency of both samples and labels. It also suggests to restrict labels to ones available at the time of training, while using samples from n-1 weeks to detect samples from the n week.

The paper [1] considered the relevance of timeline in the construction of datasets and its impact on the performance of a machine learning-based malware detectors. It shows that picking a random set of malware for training can lead to biased results. It also discusses how malware lineages could impact the performance of malware detection in the wild.

Another paper [2] further addressed the gap between the promising results that were recorded in the literature presenting ”in the lab” validation scenarios and performance of malware detectors in real-world settings. In [2] they observed F-measures dropping from over 0.9 in the lab to below 0.1 in the wild.

In this work I will change the classic model training methods in order to try and minimize the effect of covariate shifts on the performance of a deep learning malware classifier.

Regarding model training, the paper [3] claims that randomness in training, including shuffling the data, influences model performance, especially those in minority groups. Another two papers [10] [24] showed that much of the success of deep learning can be attributed to the way neural networks incorporate prior understanding of the domain in the network architecture and training.

Chapter 3

Coping with Covariate Shift in Malware-Classification

3.1 The Data

In this work I used the dataset which was published as a part of the Malicious Content Detection Platform[1] as presented in [12] and [11]. This dataset contains a full list of samples which are included in their evaluation. This list contains 1.1 million unique binaries hashes, that span over 2.5 years.

For each sample in this list, I obtained various artifacts using the VirusTotal API[2]. These include the amount of malicious detections a sample have raised across the AV engines, the behavioral information of that sample and the earliest

[1] http://secml.cs.berkeley.edu/detection_platform.

[2] https://developers.virustotal.com/reference/overview.

time the sample was submitted to VirusTotal.

For malicious samples I used the samples who raised at least 70% malicious detections across the AV engines. For benign samples I used those who raised less then 10% malicious detections across the AV engines.

In addition, I analyzed the behavioral information of each sample and extracted a sequence of the syscalls which were executed during the dynamic analysis, sorted by their time of execution. Only the samples with a minimum sequence length of 500 syscalls were selected from the dataset.

With the remaining dataset of roughly 60,000 samples I then split the data into train, validation, test and Future sets. The split fixed the malicious-benign classes ratio within each set to 50%-50%. Also, the split considered the time consistency between the datasets, I.e., test samples occur later than validation samples which occur later than train samples. This is explained thoroughly in section 4.3.1. The time consideration was used to avoid optimistic improvement when training with unrealistic samples order, which inserts ”future knowledge” to the model.

All data processing, along with download links, can be found at: https://github.com/nirosen/Malware-classification-of-the-Berkeley-detection-dataset

For any issues contact me via email at Nir.Rosen@post.idc.ac.il or Nirosen99@gmail.com.

3.2 Neural Network Architectures

In this work I used deep neural network models from different architecture types. I will give a short brief of the different types and mechanisms.

A known neural network architecture is Convolutional Neural Network (CNN). A CNN consists of an input layer, hidden layers and an output layer. The hidden layers include layers that perform convolutions, followed by other layers such as pooling, normalization and fully connected layers. CNN is applicable mostly for visual imagery tasks.

Another known class of neural network architectures is the Recurrent Neural Network (RNN). RNNs have many applications such as machine translation, time series prediction and speech recognition. The connections between the RNN nodes allows it to exhibit temporal dynamic behavior, and the RNN internal state or memory is used to keep track of long-term dependencies in variable length sequences.

A famous RNN architecture is the Long short-term memory (LSTM) architectures. The LSTM is a RNN architecture which its units composed of a cell, an input gate, an output gate and a forget gate. LSTM handles the vanishing gradient problem sometimes happens in RNN training, as it units allow gradients to flow unchanged.

Another known RNN architecture is the Gated recurrent units (GRU) architecture. The GRU is a RNN architecture with gating mechanism units. GRU is similar to LSTM, which also uses a forget gate. However, GRU has fewer parameters as it lacks an output gate. Because of that, GRU is expected to show better performance when dealing with smaller datasets.

In my work I used various pooling methods for dimension reduction. Pooling layers reduce the dimensions of data by combining the outputs of neuron clusters at one layer into a single neuron in the next layer. There are two common types of pooling in popular use: max and average. Max pooling uses the maximum value of each local cluster of neurons in the feature map, while average pooling takes the average value. In the context of RNNs, another common practice is Last Pooling which considers only the last dimension of the final RNN output.

Another technique I used is the Attention mechanism. The attention mechanism mimics cognitive attention and enhances the important parts of the input data and fades out the rest. It is applicable for machine translation and a variety of other fields including computer vision.

I considered the following nine neural network architectures to test their sensitivity to covariate shifts.

1. CNN.
2. LSTM with Average Pooling.
3. LSTM with Last Pooling.
4. LSTM with Max Pooling.
5. LSTM with Attention.
6. GRU with Average Pooling.
7. GRU with Last Pooling.
8. GRU with Max Pooling.
9. GRU with Attention.

The models were fed with the sequence of syscalls of each sample and predicted the probability of a sample being malicious or benign. I trained and evaluated each model using different methods for comparison purposes. I implemented the models using the PyTorch[3] framework and trained them in Amazon Web Services[4] cloud computing platforms for GPU acceleration.

All models implementations, along with accompanying training code, evaluation code and dataset, can be found at:
https://github.com/nirosen/AntiDriftDeepMalware

For any issues contact me via email at Nir.Rosen@post.idc.ac.il or Nirosen99@gmail.com.

[3]https://pytorch.org.
[4]https://aws.amazon.com.

Chapter 4

Experimental Design

4.1 Dev-Future Split

In this work the data set is split into two main sub-sets: the Dev-set and the Future-set, when all the samples in the Future-set occur later than the Dev-set. The Future-set remains the same for all training methods and procedures I will describe from now on.

The Dev-set is then split into another three sub-sets: train, validation and test sets. Unlike the Future-set which remains constant, the sub-sets of the Dev-set can be split either randomly or with time-consideration. When the Dev-set is split with time-consideration, the test samples occur later than validation samples which occur later than train samples. This split can also be referred to as evaluation through time.

4.2 Training Procedures

I define both split and training procedures, each learns a different subset of the data and validates and regulates differently.

4.2.1 Standard Training

The standard training procedure is a known practice in which I train the model on the train-set, evaluate its performance while determining regularization hyper-parameters based on the validation-set and finally measure its performance on the test-set.

Besides the model performance measurements that are being made on the test-set, the final performance estimate relies on the performance of the model on the Future-set and that is the one that will be used for comparison purposes.

The purpose of this procedure is to represent the known practice of model training and evaluation which is being used regularly and to compare this to the double retrain procedure that I suggest.

4.2.2 Retrain

In the retrain procedure the three sets in the Dev-set are merged into two, in order to train and evaluate the model. In this procedure I retrain the model on a union of the train and validation sets and evaluate its performance while determining regularization hyper-parameters based on the test-set.

As in all of the procedures, the final performance estimate relies on the performance of the model on the Future-set and that is the one that will be used for comparison purposes.

The purpose of this procedure is to determine regularization hyper-parameters, such as learning-rate, that will be used in later execution of the double retrain procedure. In other words, this procedure does not stand on its own but rather as a first training step to the double retrain procedure.

4.2.3 Double Retrain

In the double retrain procedure I retrain the model on all the examples in the dev-set. That is a union of the train, validation and test sets. As in all of the procedures, the final performance estimate relies on the performance of the model on the Future-set and that is the one that will be used for comparison purposes.

This procedure does not evaluate its own performance in order to determine regularization hyper-parameters such as learning-rate, but rather uses parameters that were chosen on the earlier execution of the retrain procedure described above.

The idea behind it is to relay on the first retrain to learn the hyper-parameters, but only in the second retrain to fully train the model on all the data available in the present.

4.3 Training Methods

A training method includes two main components:

1. The way the data is split into subsets, I.e., how to determine which data

points are placed in each subset.

2. The order in which the data points are organized within each subset. In the training process of a deep learning model the neural network reads the input data several times, when each cycle through the full training data is called an epoch. I consider the way of constructing an epoch, I.e., the internal order of the samples inside the train set.

I defined several training methods in order to further improve the robustness of deep learning classifiers to cope with the presence of covariate shifts.

The training-methods are defined as following:

1. Randomized-Split with Randomized-Epochs (RS-RE).

   This method implements a common practice in machine learning of random-based train-test split. In addition, it uses shuffle-based epochs, which is a common practice itself in deep learning model training. Thus, this is the most basic method of deep learning model training and evaluation. It is known as unrealistic and overly optimistic for real world scenarios. The purpose of this method is to represent the common practice of model training and evaluation which is being used regularly and to compare this to the other methods I present.

2. Time-Split with Randomized-Epochs (TS-RE).

   This method implements a known practice of time based train-test split, which is often referred to as evaluation through time. As observed in the literature and mentioned in chapter 2, it is more realistic and expected to provide with better performance estimates. In addition, this method uses shuffle-based epochs, which is a common practice in deep learning model training as mentioned. The purpose of this method is to represent a practice being implemented mostly in academic works, a practice which I see as the first step of coping with covariate shift in the field of malware detection. I will use this method as a reference to the other methods I suggest here.

3. Time-Split with Randomized-Epochs and Time-Weight (TS-RE-TW).
   This method is the first improvement I will apply to the method TS-RE. It also implements the time based train-test split and shuffle-based epochs. However, in this method I calculate a time weight for each sample, which will be used to determine the importance of the sample in the training process of the model. This is described thoroughly in subsection 4.3.2.

4. Time-Split with Time-Epochs (TS-TE).
   This method is the second improvement I will apply to the method TS-RE. It also implements the time based train-test split but in this method I construct Time-consistent Epochs instead of shuffling the samples in each epoch. Changing the order of samples inside an epoch is another approach of effecting samples importance in the training process of models, more specifically deep learning models. This is also described thoroughly in subsection 4.3.3.

4.3.1 Time based Split - evaluation through time

A train-test split is a way of partitioning the samples in the data into several sets. A known practice of a train-test split is using randomization when partitioning the samples. However, in some cases this practice interferes with a realistic evaluation of classifiers.

In a situation where the samples in the data have a meaningful order or a significant occurrence time, a randomized train-test split brings in ”future-knowledge” into the train set. I.e., the classifier trains on samples which occurred before the samples that the classifier is evaluated with. Such evaluation is unrealistic and causes inflated results, as demonstrated in the results section.

I define a time-based split, in a similar way to what was suggested in [15], where sets are ordered by the occurrence time of the samples. Time-based split can also be referred to as evaluation through time.

4.3.2 Time Weighted Samples

The use of weighted samples in cases of unbalanced data, such as uneven classes ratio in the data, is a known practice in the training process of a deep learning model.

I define the time weight of a sample in equation 4.1, in which the occurrence time of a sample effects its weight in the training process of the neural network. That practice is proposed in order to give new samples a more significant effect than older ones.

$$w_{sample} = e^{-a \cdot \frac{t}{T}} \tag{4.1}$$

Where:

t = time gap to the latest sample in the set (in a resolution of seconds)

T = time gap from the first sample to the latest sample in the set

a = hyper parameter of the time weight (defaults to zero)

Given the time weights of the samples in the batch, the weighted batch loss, $L_{weighted}$, is calculated as following:

$$L_{weighted} = L \cdot W$$

Where L is the current batch loss and W is a vector containing the weights of the samples in the batch.

4.3.3 Time-consistent Epochs

In the training process of a deep learning model the neural network reads the input data several times, when each cycle through the full training data is called an epoch. A known practice of constructing epochs uses randomization, in which one feeds a neural network the data in a shuffled pattern in each epoch.

I define time consistent epochs, where the train set is ordered by the occurrence time of the samples and is being fed to the neural network in a specific time consistent order, rather than a shuffled pattern each epoch.

This practice is suggested since neural networks have an inherent inductive bias that causes them to assign more weight to recent examples over old ones. This type of bias can be addressed by adding regularization or modifying the learning rate, but I wish to leverage it to improve robustness in the face of covariate shifts.

When epochs are being ordered by the occurrence time of the samples, the last samples that are fed to the model have a newer occurrence time than the first ones. This creates an inductive bias towards patterns that appear in up to date samples and helps the model to generalize to the latest patterns.

4.4 Metrics

I evaluated the performance of the models in this work using the ROC-AUC metric.

The Receiver Operating Characteristic (ROC) curve is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters: True Positive Rate and False Positive Rate.

True Positive Rate (TPR), also referred to as recall, is defined as follows:

$TPR = \frac{TP}{TP+FN}$

False Positive Rate (FPR) is defined as follows:

$FPR = \frac{FP}{FP+TN}$

The Area Under the ROC Curve (ROC-AUC), referred to as AUC, measures the entire two-dimensional area underneath the entire ROC curve from (0,0) to (1,1). It provides an aggregate measure of performance across all possible classification thresholds.

ROC curves can have similar AUC values and still be significantly different. In order to compare the performance of two models I use the Confidence Interval of the AUC (CI-AUC), which is computed with Delong's method [7]. I use this method since it does not require bootstrapping, which is crucial for evaluating the training methods I present. However, it also can not handle Partial AUC (pAUC), which is often used in the malware detection domain for measuring AUC at low FPR. Thus, for models comparison I will use the full ROC CI-AUC.

Chapter 5

Experimental Results

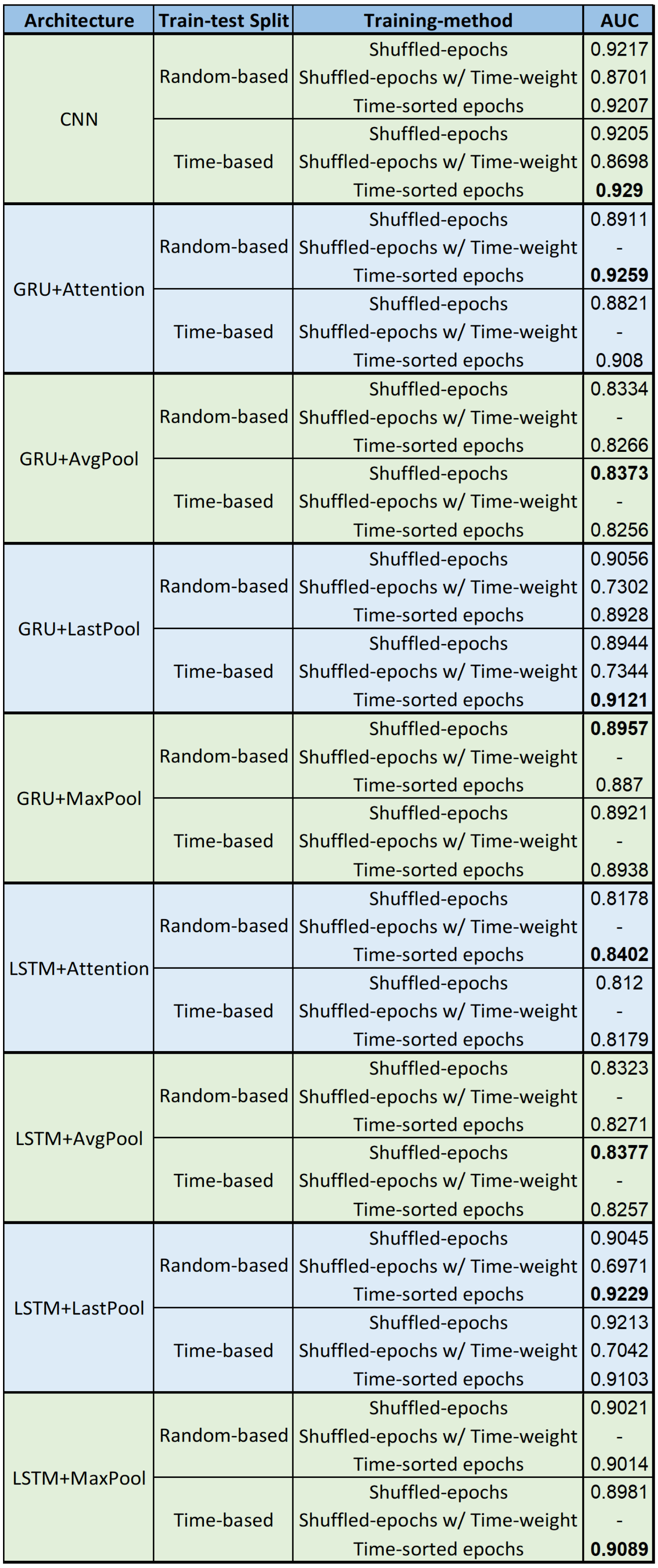
I tested the methods described above on nine different architectures and present the final AUC score. All models were trained using the Double-Retrain procedure on the same Dev-set and evaluated on the same Future-set. Results are shown in Table 5.1, with the best performing training method marked for each architecture.

The comparisons of the methods are described thoroughly in the hypotheses section 6.1. The raised hypotheses were verified using the CI-AUC as explained in the metrics section 4.4. I derived the following conclusions from the hypotheses that showed consistency across the network architectures.

I verified that Time-Split based methods, often referred to as evaluation through time, provides good performance estimates in the presence of covariate shifts. This was verified as explained in listing 4 of the hypotheses section 6.1. Out of the nine architectures I tested, the only one that did not verify this claim was the LSTM with Attention, as can be seen in appendix A.

The new training method I suggested utilizes both Time based Split and Time sorted Epochs. I observed that in addition to preserving good performance estimates, my method also improved the actual performance of the model for three out of the nine network architectures as can be seen in Table 5.1. The three architectures that performed better with my method were the GRU with Last Pooling, the LSTM with Max Pooling and the CNN. The CNN architecture trained with my method was also the best performing model which achieved the highest AUC score of 92.9%.

In my experiments I also tested the training method that involves Time-Weight. However, this method performed much worst than the base method so I did not test it on all the architectures due to limited resources.



Architecture	Train-test Split	Training-method	AUC
CNN	Random-based	Shuffled-epochs	0.9217
		Shuffled-epochs w/ Time-weight	0.8701
		Time-sorted epochs	0.9207
	Time-based	Shuffled-epochs	0.9205
		Shuffled-epochs w/ Time-weight	0.8698
		Time-sorted epochs	**0.929**
GRU+Attention	Random-based	Shuffled-epochs	0.8911
		Shuffled-epochs w/ Time-weight	-
		Time-sorted epochs	**0.9259**
	Time-based	Shuffled-epochs	0.8821
		Shuffled-epochs w/ Time-weight	-
		Time-sorted epochs	0.908
GRU+AvgPool	Random-based	Shuffled-epochs	0.8334
		Shuffled-epochs w/ Time-weight	-
		Time-sorted epochs	0.8266
	Time-based	Shuffled-epochs	**0.8373**
		Shuffled-epochs w/ Time-weight	-
		Time-sorted epochs	0.8256
GRU+LastPool	Random-based	Shuffled-epochs	0.9056
		Shuffled-epochs w/ Time-weight	0.7302
		Time-sorted epochs	0.8928
	Time-based	Shuffled-epochs	0.8944
		Shuffled-epochs w/ Time-weight	0.7344
		Time-sorted epochs	**0.9121**
GRU+MaxPool	Random-based	Shuffled-epochs	**0.8957**
		Shuffled-epochs w/ Time-weight	-
		Time-sorted epochs	0.887
	Time-based	Shuffled-epochs	0.8921
		Shuffled-epochs w/ Time-weight	-
		Time-sorted epochs	0.8938
LSTM+Attention	Random-based	Shuffled-epochs	0.8178
		Shuffled-epochs w/ Time-weight	-
		Time-sorted epochs	**0.8402**
	Time-based	Shuffled-epochs	0.812
		Shuffled-epochs w/ Time-weight	-
		Time-sorted epochs	0.8179
LSTM+AvgPool	Random-based	Shuffled-epochs	0.8323
		Shuffled-epochs w/ Time-weight	-
		Time-sorted epochs	0.8271
	Time-based	Shuffled-epochs	**0.8377**
		Shuffled-epochs w/ Time-weight	-
		Time-sorted epochs	0.8257
LSTM+LastPool	Random-based	Shuffled-epochs	0.9045
		Shuffled-epochs w/ Time-weight	0.6971
		Time-sorted epochs	**0.9229**
	Time-based	Shuffled-epochs	0.9213
		Shuffled-epochs w/ Time-weight	0.7042
		Time-sorted epochs	0.9103
LSTM+MaxPool	Random-based	Shuffled-epochs	0.9021
		Shuffled-epochs w/ Time-weight	-
		Time-sorted epochs	0.9014
	Time-based	Shuffled-epochs	0.8981
		Shuffled-epochs w/ Time-weight	-
		Time-sorted epochs	**0.9089**

Figure 5.1: Results table.

Chapter 6

Discussion

6.1 Hypotheses

In my work I raised hypotheses about the performance of the training methods and tested them on the different architectures. I checked the correctness of the hypotheses in appendix A with the performance of each model compared using the CI-AUC.

I raised the following hypotheses regarding the training methods and procedures detailed in sections 4.2 and 4.3:

1. I argue that the RSRE method will overtake the other methods (TSRE/TSTE) when evaluating the Future-set using the Standard Training procedure. This is because the RSRE method that contains Random based Split inserts newer samples to the train set than the Time based Split methods.

2. (a) I argue that the TSRE method will overtake the RSRE method when evaluating the Future-set using the Double Retrain procedure.

   (b) I argue that the TSTE method will overtake the RSRE method when evaluating the Future-set using the Double Retrain procedure.

3. I argue that the TSTE method will overtake the TSRE method when evaluating the Future-set using the Double Retrain procedure.

4. I argue that RSRE method is overly optimistic when predicting its own performance. Specifically, I argue that TSRE and TSTE methods are more realistic in predicting their performance.

   (a) I argue that RSRE will show a significant gap in the following evaluations:

      i. The gap between:

         A. Evaluating the Test-set using the Standard Training procedure.

         B. Evaluating the Future-set using the Standard Training procedure.

      ii. The gap between:

         A. Evaluating the Test-set using the Single Retrain procedure.

         B. Evaluating the Future-set using the Single Retrain procedure

      iii. The gap between:

         A. Evaluating the Test-set using the Standard Training procedure.

         B. Evaluating the Future-set using the Double Retrain procedure.

      iv. The gap between:

        A. Evaluating the Test-set using the Single Retrain procedure.

        B. Evaluating the Future-set using the Double Retrain procedure.

   (b) I argue that TSRE and TSTE methods will show a significantly smaller gap (Compared to all of the above) in the following evaluation:

      i. The gap between:

      ii. Evaluating the Test-set using the Single Retrain procedure.

      iii. Evaluating the Future-set using the Double Retrain procedure

5. I argue that RSTE method will overtake the RSRE method when evaluating the Future-set using the Double Retrain procedure.

See appendix A for the results of the hypotheses.

6.2 Future Directions

In this work I tried to change the training method of a deep learning model in order to cope with covariate shift in malware classifiers. In my future work, I intend to continue with the focus on the training phase of a model using the active learning approach. Using active learning, I will be able to choose the samples that will be fed into the model according to the performance of the model. E.g., if a model performs poorly when tested on newly dated samples I will construct the next training batch with up to date samples. I conjecture that active learning could have a major benefit of coping with covariate shift in malware detection.

References

[1] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Are your training datasets yet relevant? In International Symposium on Engineering Secure Software and Systems, pages 51–67. Springer, 2015.

[2] Kevin Allix, Tegawendé François D Assise Bissyande, Quentin Jerome, Jacques Klein, Yves Le Traon, et al. Empirical assessment of machine learning-based malware detectors for android: Measuring the gap between in-the-lab and in-the-wild validation scenarios. Empirical Software Engineering, pages 1–29, 2014.

[3] Silvio Amir, Jan-Willem van de Meent, and Byron C. Wallace. On the impact of random seeds on the fairness of clinical classifiers, 2021.

[4] Federico Barbero, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. Transcending transcend: Revisiting malware classification with conformal evaluation. arXiv e-prints, pages arXiv–2010, 2020.

[5] Ishita Basu, Nidhi Sinha, Diksha Bhagat, and Saptarsi Goswami. Malware detection based on source data using data mining: A survey. American Journal of Advanced Computing, 3(01):18–37, 2016.

[6] Geoff Chappell. Win32 Programming and Kernel-Mode Windows, 2019. https://www.geoffchappell.com/studies/windows/win32/kernel32/api/index.htm, https://www.geoffchappell.com/studies/windows/km/ntoskrnl/api/index.htm.

[7] E DeLong, D DeLong, and D Clarke-Pearson. Comparing the areas under two or more correlated receiver operating characteristic curves: a nonparametric approach. biome trics 1988; 44: 837-45. Am J Nephrol, 45:400–8, 2017.

[8] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In 26th {USENIX} Security Symposium ({USENIX} Security 17), pages 625–642, 2017.

[9] Chan Woo Kim. Ntmaldetect: A machine learning approach to malware detection using native api system calls. arXiv preprint arXiv:1802.05412, 2018.

[10] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. nature, 521(7553):436–444, 2015.

[11] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, et al. Reviewer integration and performance measurement for malware detection. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pages 122–141. Springer, 2016.

[12] Bradley Miller. Scalable Platform for Malicious Content Detection

Integrating Machine Learning and Manual Review. PhD thesis, EECS Department, University of California, Berkeley, Aug 2015.

[13] Nikhil Muralidhar, Sathappah Muthiah, Patrick Butler, Manish Jain, Yu Yu, Katy Burne, Weipeng Li, David Jones, Prakash Arunachalam, Hays 'Skip' McCormick, and Naren Ramakrishnan. Using antipatterns to avoid mlops mistakes, 2021.

[14] Yaniv Ovadia, Emily Fertig, Jie Ren, Zachary Nado, D Sculley, Sebastian Nowozin, Joshua V. Dillon, Balaji Lakshminarayanan, and Jasper Snoek. Can you trust your model's uncertainty? evaluating predictive uncertainty under dataset shift, 2019.

[15] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In 28th {USENIX} Security Symposium ({USENIX} Security 19), pages 729–746, 2019.

[16] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. arXiv preprint arXiv:1802.10135, 2018.

[17] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Generic black-box end-to-end attack against state of the art api call based malware classifiers. In International Symposium on Research in Attacks, Intrusions, and Defenses, pages 490–510. Springer, 2018.

[18] Asaf Shabtai, Robert Moskovitch, Clint Feher, Shlomi Dolev, and Yuval Elovici. Detecting unknown malicious code by applying classification tech-

niques on opcode patterns. Security Informatics, 1(1):1, 2012.

[19] Muazzam Siddiqui, Morgan C Wang, and Joohan Lee. A survey of data mining techniques for malware detection using file features. In Proceedings of the 46th annual southeast regional conference on xx, pages 509–510, 2008.

[20] Michael Smith, Joey Ingram, Christopher Lamb, Timothy Draelos, Justin Doak, James Aimone, and Conrad James. Dynamic analysis of executables to detect and characterize malware. In 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), pages 16–22. IEEE, 2018.

[21] Alireza Souri and Rahil Hosseini. A state-of-the-art survey of malware detection approaches using data mining techniques. Human-centric Computing and Information Sciences, 8(1):3, 2018.

[22] Jack W Stokes, De Wang, Mady Marinescu, Marc Marino, and Brian Bussone. Attack and defense of dynamic analysis-based, adversarial neural malware classification models. arXiv preprint arXiv:1712.05919, 2017.

[23] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. Computers & Security, 81:123–147, 2019.

[24] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Deep image prior. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 9446–9454, 2018.

[25] Yanfang Ye, Tao Li, Donald Adjeroh, and S Sitharama Iyengar. A survey on

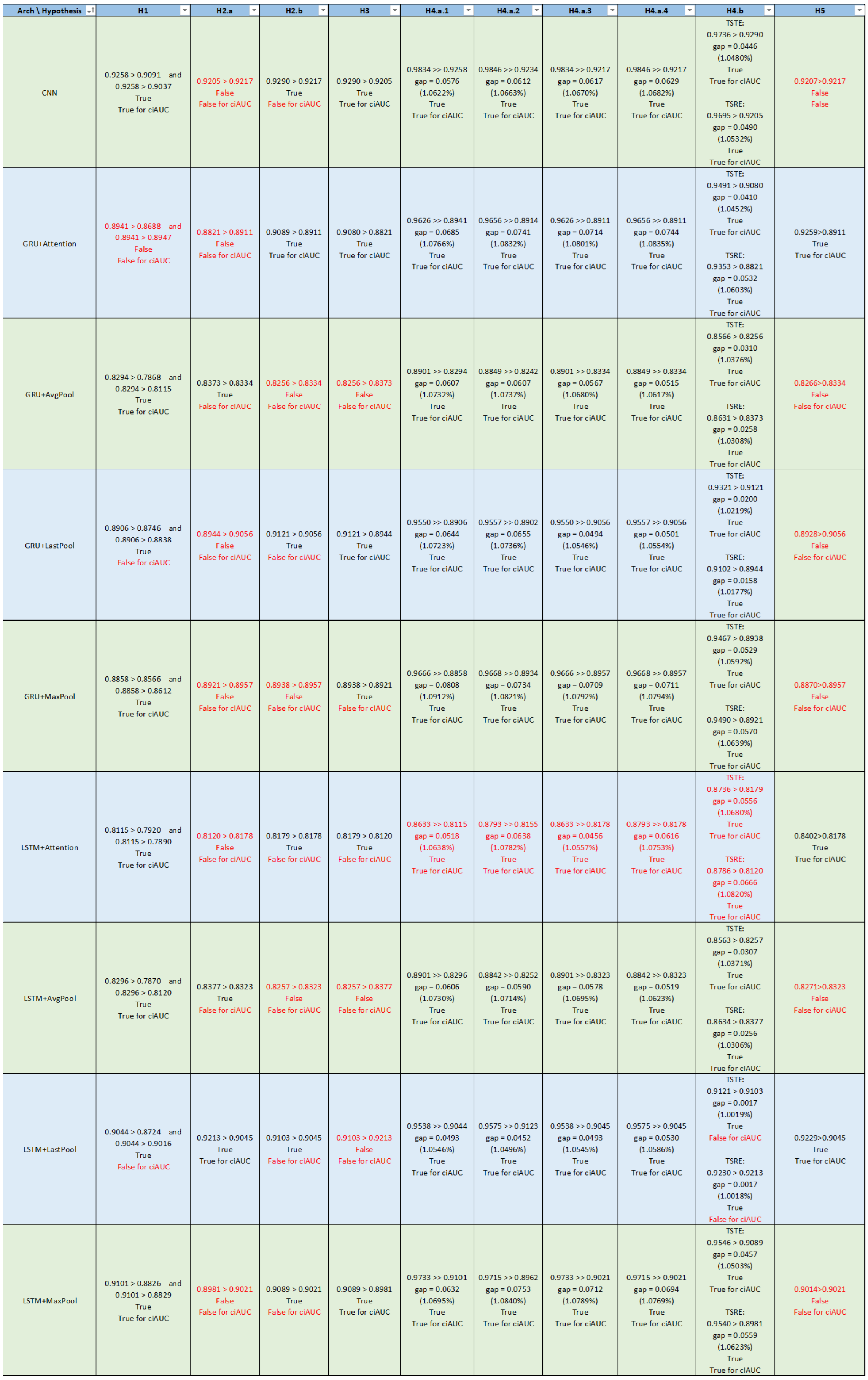
malware detection using data mining techniques. ACM Computing Surveys (CSUR), 50(3):1–40, 2017.

[26] Hongwei Zhao, Mingzhao Li, Taiqi Wu, and Fei Yang. Evaluation of supervised machine learning techniques for dynamic malware detection. International Journal of Computational Intelligence Systems, 11(1):1153–1169, 2018.

Appendix A

Hypotheses Results

Table A.1 contains detailed results of the hypotheses I raised in my work. I tested nine different architectures and verified the hypotheses using the CI-AUC as explained in the metrics section 4.4. The comparisons that were proved wrong are labeled as ”False” and are highlighted in the table. Notice that some hypotheses passed the basic AUC comparison but failed the CI-AUC comparison, thus considered as false ones.



Arch \ Hypothesis	H1	H2.a	H2.b	H3	H4.a.1	H4.a.2	H4.a.3	H4.a.4	H4.b	H5
CNN	0.9258 > 0.9091 and 0.9258 > 0.9037 True True for ciAUC	0.9205 > 0.9217 False False for ciAUC	0.9290 > 0.9217 True False for ciAUC	0.9290 > 0.9205 True True for ciAUC	0.9834 >> 0.9258 gap = 0.0576 (1.0622%) True True for ciAUC	0.9846 >> 0.9234 gap = 0.0612 (1.0663%) True True for ciAUC	0.9834 >> 0.9217 gap = 0.0617 (1.0670%) True True for ciAUC	0.9846 >> 0.9217 gap = 0.0629 (1.0682%) True True for ciAUC	TSTE: 0.9736 > 0.9290 gap = 0.0446 (1.0480%) True True for ciAUC TSRE: 0.9695 > 0.9205 gap = 0.0490 (1.0532%) True True for ciAUC	0.9207>0.9217 False False
GRU+Attention	0.8941 > 0.8688 and 0.8941 > 0.8947 False False for ciAUC	0.8821 > 0.8911 False False for ciAUC	0.9089 > 0.8911 True True for ciAUC	0.9080 > 0.8821 True True for ciAUC	0.9626 >> 0.8941 gap = 0.0685 (1.0766%) True True for ciAUC	0.9656 >> 0.8914 gap = 0.0741 (1.0832%) True True for ciAUC	0.9626 >> 0.8911 gap = 0.0714 (1.0801%) True True for ciAUC	0.9656 >> 0.8911 gap = 0.0744 (1.0835%) True True for ciAUC	TSTE: 0.9491 > 0.9080 gap = 0.0410 (1.0452%) True True for ciAUC TSRE: 0.9353 > 0.8821 gap = 0.0532 (1.0603%) True True for ciAUC	0.9259>0.8911 True True for ciAUC
GRU+AvgPool	0.8294 > 0.7868 and 0.8294 > 0.8115 True True for ciAUC	0.8373 > 0.8334 True False for ciAUC	0.8256 > 0.8334 False False for ciAUC	0.8256 > 0.8373 False False for ciAUC	0.8901 >> 0.8294 gap = 0.0607 (1.0732%) True True for ciAUC	0.8849 >> 0.8242 gap = 0.0607 (1.0737%) True True for ciAUC	0.8901 >> 0.8334 gap = 0.0567 (1.0680%) True True for ciAUC	0.8849 >> 0.8334 gap = 0.0515 (1.0617%) True True for ciAUC	TSTE: 0.8566 > 0.8256 gap = 0.0310 (1.0376%) True True for ciAUC TSRE: 0.8631 > 0.8373 gap = 0.0258 (1.0308%) True True for ciAUC	0.8266>0.8334 False False for ciAUC
GRU+LastPool	0.8906 > 0.8746 and 0.8906 > 0.8838 True False for ciAUC	0.8944 > 0.9056 False False for ciAUC	0.9121 > 0.9056 True False for ciAUC	0.9121 > 0.8944 True True for ciAUC	0.9550 >> 0.8906 gap = 0.0644 (1.0723%) True True for ciAUC	0.9557 >> 0.8902 gap = 0.0655 (1.0736%) True True for ciAUC	0.9550 >> 0.9056 gap = 0.0494 (1.0546%) True True for ciAUC	0.9557 >> 0.9056 gap = 0.0501 (1.0554%) True True for ciAUC	TSTE: 0.9321 > 0.9121 gap = 0.0200 (1.0219%) True True for ciAUC TSRE: 0.9102 > 0.8944 gap = 0.0158 (1.0177%) True True for ciAUC	0.8928>0.9056 False False for ciAUC
GRU+MaxPool	0.8858 > 0.8566 and 0.8858 > 0.8612 True True for ciAUC	0.8921 > 0.8957 False False for ciAUC	0.8938 > 0.8957 False False for ciAUC	0.8938 > 0.8921 True False for ciAUC	0.9666 >> 0.8858 gap = 0.0808 (1.0912%) True True for ciAUC	0.9668 >> 0.8934 gap = 0.0734 (1.0821%) True True for ciAUC	0.9666 >> 0.8957 gap = 0.0709 (1.0792%) True True for ciAUC	0.9668 >> 0.8957 gap = 0.0711 (1.0794%) True True for ciAUC	TSTE: 0.9467 > 0.8938 gap = 0.0529 (1.0592%) True True for ciAUC TSRE: 0.9490 > 0.8921 gap = 0.0570 (1.0639%) True True for ciAUC	0.8870>0.8957 False False for ciAUC
LSTM+Attention	0.8115 > 0.7920 and 0.8115 > 0.7890 True True for ciAUC	0.8120 > 0.8178 False False for ciAUC	0.8179 > 0.8178 True False for ciAUC	0.8179 > 0.8120 True False for ciAUC	0.8633 >> 0.8115 gap = 0.0518 (1.0638%) True True for ciAUC	0.8793 >> 0.8155 gap = 0.0638 (1.0782%) True True for ciAUC	0.8633 >> 0.8178 gap = 0.0456 (1.0557%) True True for ciAUC	0.8793 >> 0.8178 gap = 0.0616 (1.0753%) True True for ciAUC	TSTE: 0.8736 > 0.8179 gap = 0.0556 (1.0680%) True True for ciAUC TSRE: 0.8786 > 0.8120 gap = 0.0666 (1.0820%) True True for ciAUC	0.8402>0.8178 True True for ciAUC
LSTM+AvgPool	0.8296 > 0.7870 and 0.8296 > 0.8120 True True for ciAUC	0.8377 > 0.8323 True False for ciAUC	0.8257 > 0.8323 False False for ciAUC	0.8257 > 0.8377 False False for ciAUC	0.8901 >> 0.8296 gap = 0.0606 (1.0730%) True True for ciAUC	0.8842 >> 0.8252 gap = 0.0590 (1.0714%) True True for ciAUC	0.8901 >> 0.8323 gap = 0.0578 (1.0695%) True True for ciAUC	0.8842 >> 0.8323 gap = 0.0519 (1.0623%) True True for ciAUC	TSTE: 0.8563 > 0.8257 gap = 0.0307 (1.0371%) True True for ciAUC TSRE: 0.8634 > 0.8377 gap = 0.0256 (1.0306%) True True for ciAUC	0.8271>0.8323 False False for ciAUC
LSTM+LastPool	0.9044 > 0.8724 and 0.9044 > 0.9016 True False for ciAUC	0.9213 > 0.9045 True True for ciAUC	0.9103 > 0.9045 True False for ciAUC	0.9103 > 0.9213 False False for ciAUC	0.9538 >> 0.9044 gap = 0.0493 (1.0546%) True True for ciAUC	0.9575 >> 0.9123 gap = 0.0452 (1.0496%) True True for ciAUC	0.9538 >> 0.9045 gap = 0.0493 (1.0545%) True True for ciAUC	0.9575 >> 0.9045 gap = 0.0530 (1.0586%) True True for ciAUC	TSTE: 0.9121 > 0.9103 gap = 0.0017 (1.0019%) True False for ciAUC TSRE: 0.9230 > 0.9213 gap = 0.0017 (1.0018%) True False for ciAUC	0.9229>0.9045 True True for ciAUC
LSTM+MaxPool	0.9101 > 0.8826 and 0.9101 > 0.8829 True True for ciAUC	0.8981 > 0.9021 False False for ciAUC	0.9089 > 0.9021 True False for ciAUC	0.9089 > 0.8981 True True for ciAUC	0.9733 >> 0.9101 gap = 0.0632 (1.0695%) True True for ciAUC	0.9715 >> 0.8962 gap = 0.0753 (1.0840%) True True for ciAUC	0.9733 >> 0.9021 gap = 0.0712 (1.0789%) True True for ciAUC	0.9715 >> 0.9021 gap = 0.0694 (1.0769%) True True for ciAUC	TSTE: 0.9546 > 0.9089 gap = 0.0457 (1.0503%) True True for ciAUC TSRE: 0.9540 > 0.8981 gap = 0.0559 (1.0623%) True True for ciAUC	0.9014>0.9021 False False for ciAUC

Figure A.1: Hypotheses results table.

תקציר

איתור תוכנות זדוניות הינו אחד האתגרים הגדולים ביותר בתחום אבטחת המידע. כיום ישנו שימוש אינטנסיבי של למידה עמוקה על-מנת לאתר או לסווג תוכנות זדוניות. עם זאת, תוכנות זדוניות משתנות במהירות על-מנת להימנע מאיתור, עובדה אשר הופכת היסט בנתונים לאתגר מרכזי באיתור תוכנות זדוניות. המושג היסט נתונים מתייחס לשינויים בהתפלגות המידע. הדבר מעלה אתגרים רבים באימון ובהערכה של מסווגים מבוססי למידת מכונה בתחום איתור התוכנות הזדוניות, וכן מגביל את הרלוונטיות של מסווגים בטווח הארוך.

אני מציג שיטה ידועה של חלוקת הדאטה לאימון ומבחן אשר מבוססת זמן, המכונה לעיתים הערכה לאורך זמן. אני מדגים שיטה זו בתחום איתור התוכנות זדוניות ומראה כי היא מספקת הערכת ביצועים טובה בעת היסט בנתונים.

בנוסף, אני מציג שיטה חדשה שמטרתה לאמן ולהעריך מודלים של למידה עמוקה על-מנת להפכם לחסינים בפני היסט בנתונים וכן לספק הערכת ביצועים טובה. דבר זה מושג דרך שינוי שיטת האימון ללא שינויים לארכיטקטורת הרשת.

השיטה שלי משנה את תהליך האימון הקלאסי בשתי דרכים מרכזיות. ראשית, אני נמנע מערבוב הדוגמאות בסט האימון וכחלופה מזין אותן לרשת ממוינות לפי זמן. הדבר מייצר הטיה אינדוקטיבית לכיוון דפוסים המופיעים בדוגמאות מאוחרות יותר ומסייע למודל לבצע הכללה של הדפוסים העדכניים ביותר.

שנית, אני מפצל את המידע לסטים של אימון, הערכה ומבחן לאורך זמן ומאמן את המודל מחדש פעמיים. האימון מחדש הראשון מיועד לאופטימיזציה של היפר פרמטרים בהתבסס על תפקוד סט המבחן. האימון מחדש השני מיועד להכשרה של המודל הסופי על כלל המידע, כולל הדוגמאות האחרונות. לצורך הערכה של התוצאות שמרתי בצד סט עתידי, אשר מנותק משאר הסטים ומכיל את הדוגמאות העדכניות ביותר.

בחנתי את השיטה שלי על תשע ארכיטקטורות למידה עמוקה שונות וחישבתי את מרווחי הביטחון של ה-AUC על-מנת להוכיח הבדל סטטיסטי עבור שיפור בביצועים. בנוסף לכך שהשיטה סיפקה הערכת ביצועים טובה, הראיתי שהשיטה אף שיפרה את הביצועים בפועל של שלוש מתוך תשע ארכיטקטורות הרשת אותן ניסיתי.

עבודה זו בוצעה בהדרכתו של דר' לירון אלרהנד מאוניברסיטת תל אביב ודר' אילן גרונאו מבי"ס אפי ארזי למדעי המחשב, אוניברסיטת רייכמן.

אוניברסיטת רייכמן
בית-ספר אפי ארזי למדעי המחשב
התכנית לתואר שני (M.Sc.)

התמודדות עם היסט נתונים במסווגי תוכנות זדוניות

מאת
ניר רוזן

פרויקט גמר, מוגש כחלק מהדרישות לשם קבלת תואר מוסמך M.Sc.,
בית ספר אפי ארזי למדעי המחשב, אוניברסיטת רייכמן

דצמבר 2021