IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 18 January 2024, accepted 9 May 2024, date of publication 15 May 2024, date of current version 28 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3401627

RESEARCH ARTICLE

TRConv: Multi-Platform Malware Classification via Target Regulated Convolutions

ALPER EGITMEN[1], ALI GOKHAN YAVUZ[2], AND SIRMA YAVUZ[3], (Member, IEEE)
[1]Crossfirelab Software Inc., Esenler, 34220 Istanbul, Turkey
[2]Department of Computer Engineering, Turkish-German University, 34820 Istanbul, Turkey
[3]Department of Computer Engineering, Yıldız Technical University, 34349 Istanbul, Turkey

Corresponding author: Alper Egitmen (alper.egitmen@crossfirelab.com)

This work was supported in part by Yıldız Technical University (YTU) Scientific Research Projects Coordination Unit (BAPK) under Project FBA-2022-4904.

ABSTRACT Malware is an important threat to digital workflow. Traditional malware modeling approaches focused on using hand-crafted features while recent approaches proved the necessity of using learning based methodologies. In this paper, we propose a novel opcode based methodology that additionally learns multiple behavioral target variables to effectively regulate and guide the static malware classification. Our methodology shows that introduction of previously extracted malware behavior-related target variables immediately improve binary malware classification performance in both Android and Windows platforms. The contributions of our methodology has been extensively validated on the AMDArgus and the MOTIF dataset. Mean classification accuracy and F1 scores suggest that our model is robust against random opcode injection attacks compared to other convolution based architectures.

INDEX TERMS Convolutional network, opcode length, malware and benign, malware behaviour analysis.

I. INTRODUCTION

Malware and disruptions to digital workflows pose significant socio-economic threats. Both at the corporate and individual levels, systems face persistent and indiscriminate cyber attacks, leading to consequences such as resource exploitation, service malfunction, unavailability, and theft. Compared to few decades ago, today's world is populated with Android based systems such as mobile devices, internet of things (IOT) devices and vehicles. This widespread usage of Android systems along with increased connectivity and network transfer rates contributed number of malicious malware and users' rate of exposure. Now, it is established that the problems arising from this inclusiveness and accessibility is only surmountable with the development of an accurate automated analysis. However, accuracy of such an analysis necessitates novel techniques and reasonable approximations for rapid inference to ensure maximum protection. Thus, rapid identification of large volumes of cross-platform malware with such automated methods with minimal human intervention is still under exploration.

The associate editor coordinating the review of this manuscript and approving it for publication was Giovanni Pau.

Traditional approaches to malware classification relied on using hand-crafted features, while modern approaches proved the necessity of using learning based methodologies to achieve generality. Many of these learning based approaches use neural architectures. Malware analysis is usually categorized by whether the analysis is offline or real-time. Offline analysis known as static analysis relies on the metadata and binary of the program, without access to the runtime behavior of the malware. In contrast, dynamic analysis requires a monitored environment to observe and investigate the program's behavior. We perform static analysis and focus on the well-learnability of malware.

In this paper, we propose a new methodology to address static malware classification task. Although there are various approaches modeling the malware classification task, a model which effectively regularizes the opcode sequences, and shows satisfying performance in a wide setting is missing. In this paper, we address this problem and propose a novel convolutional architecture. Briefly put, the contributions of this paper are:

- Faster and more reliable training by incorporation of opcode pretraining to a convolutional architecture.

- Formulation of malware classification as joint learning of multiple related but weakly reliable target variables.
- Unlike many existing approaches, we validate the efficacy of our methodology on both Windows and Android architectures.
- Evaluation of our model's performance in various real-world scenarios.

II. RELATED WORK

A. MALWARE ANALYSIS APPROACHES

Malware detection methods perform the analysis either in a static or dynamic fashion. In static analysis, the malware is analyzed only by assuming access to malware's binary, and occasionally meta-data based access statistics. Thus, one has no knowledge of its real-time behavior. In this conventional method of malware analysis, a disassembler tool is used to obtain the assembly code of the malware. Then, input modalities can be formed on different levels of representation such as raw-bytes or opcodes, along with high level features such as assembly code flow. For instance, Jiang et al. [1] categorize opcodes as sensitive and non-sensitive by the help of API calls and sensitive metadata before feeding them to classifiers.

Dynamic analysis is a more extensive analysis of malware. In dynamic analysis, the malware is analyzed while it runs, meaning that its behavior can be monitored in real-time and its runtime statistics can be immediately taken into account. With dynamic analysis, one obtains useful new features such as network usage, file accesses, system calls and runtime metrics. An example is found in Catal et al.'s work [2] where they perform a multi stage dynamical analysis by using API call graphs, followed by Node2Vec to create node representations which is fed to a graph neural network models, namely Generative Adversarial Networks (GAN) and Graph Convolutional Network (GCN).

Some works follow a joint approach to increase the modeling power. Aktas and Sen [3] propose a hybrid analysis where static and dynamic features are aggregated together.

B. INPUT MODALITIES IN MALWARE ANALYSIS

Both on static or dynamic approaches, different input modalities are used. Yan et al. [4] present an opcode-based analysis, extracting opcodes from Android Package Kit (APK) files. Recurring opcode patterns are further filtered to increase the signal to noise ratio of the opcode sequences. To address the challenge posed by varying opcode sequence lengths in APK files, they define method blocks to normalize sequence lengths for proper training, especially for Long Short-Term Memory (LSTM) autoencoder-based approaches.

These problems are encountered on Windows malware as well. Anderson and Roth [5] propose EMBER, which captures common statistical features of Portable Executable (PE) files, determined by an analysis of human experts. Raju and Wang [6] perform a Malware Capabilities (CAPA) labeling-based analysis on PE files, extracting predefined identifiers such as API imports, parsing errors, entropy and byte distributions by the help of subject matter experts. These extracted features are further complemented by EMBER features. Kadri et al. [7] show the increased benefit of using opcode representations over raw bytes modeling. Considering risks of raw byte modeling into account, Vinayakumar et al. [8] use combination of raw bytes and EMBER features which are extracted from given malware a priori. Pektaş and Acarman [9] extract opcodes and system call graph features in a dynamic-analysis scenario. Saxe and Berlin [10] use a combination of features to model malware classification task. They use contextual byte features, metadata features of PE and byte entropy histograms.

Contrary to learning opcode vectors, Raff et al. [11] propose using N-grams for modeling opcode sequences. They use up to 6 grams to represent the files. Although these approaches are quite robust when they saw a predefined pattern, there are two key limitations that prevents these solutions to scale up. First, higher order N-gram modeling suffers from memory limitations and the need for developing additional mechanisms to deal with the inherent data sparsity. Thus, we do not utilize any traditional N-gram features, and stick with fully learned features in this paper.

There are variations in how researchers implement an opcode based malware representation. Ghezelbigloo and VafaeiJahan [12] use so called role-opcodes, an opcode grouping approach. As a result, compared to the standard opcode representations, they end up in a much smaller vocabulary space. Zhang et al. [13] incorporate system call sequences along with opcodes to perform a static analysis. In dynamic analysis, Yesir and Soğukpinar work [14] constitutes and example where he incorporates API call sequences. Lucas et al. [15] address the sensitivity of MalConv [16] and AvastNet to random alterations to opcode sequences, suggesting that such simple alterations can lead to very high evasion rates.

Motivated by the recent successes in image classification domain, there has been a tendency to represent the raw malware file as a 2D image [17], [18], [19], [20]. For instance, Xu et al. propose the SeqNet architecture [21] which transforms the input malware into an image by introducing techniques to minimize resampling and edge losses during this conversion.

Bensaud et al. [22] consider malware file as a 2D image with pixel intensities byte values and learn the class of image with well-known networks such as Inception, VGG16, Resnet50. Similar modeling has been done with adversarial networks [17]. Even simple CNNs also showed extremely high accuracy [23]. Usually, these works form an image using raw malware bytes. In contrast to this, in Xing et al's work [24] authors first extract the opcodes, and then form a malware image representation by viewing opcodes as pixels.

Although transforming the malware to a 2D image enables using a suite of standard image classification models, there is no clear evidence why malware should be treated as an image.

A byte in a malware is highly dependent on its location in the code. However, in images a pixel always captures a color or intensity information. By representing malware 2D, we add some horizontal connections which does not necessarily hold, there should be some constraints imposed to penalize number of such connections. In this paper, we do not explore such directions, and stick with the non-spatial, unidimensional representation of the malware.

C. MODEL ARCHITECTURES

Since the problem of malware modeling has many different aspects, a large variety of complex architectures have been used for addressing this task. A subset of approaches suggest incorporating large language models (LLMs). For instance, Long Short-Term Memory (LSTM) networks are employed in Dang et al.'s work [25]. Balikcioglu et al. [26] also propose a LSTM method to model Android instructions. Unlike a standard approach that only represent features at the lowest level, such as individual machine instructions, they further bag these instruction features into function-wise and class-wise fashion. Demirci et al. [27] offer several approaches for handling android opcodes and focus on systematic comparison of pre-trained fine tuned transformer models including Generative Pre-trained Transformer 2 (GPT2), distilBERT and Stacked Bidirectional LSTM (BiLSTM), providing a thorough comparison. Ding et al. [28] address the multi-class problem particularly when samples exhibit a high imbalance ratio. Their model enjoys a self-attention mechanism to efficiently localize relevant regions in the feature representation. This approach is further extended using additional strategies such as selecting top opcodes and sequence truncation.

Although convolutional approaches are effective for malware modeling, under some circumstances they are vulnerable to attacks. In this branch of work, Demetrio et al. [29] use a CNN based approach and analyzes the adverserial properties of deep learning methods to model the malware. Kolosnjaji et al. [30] highlight weaknesses of simple convolutional approaches to malware classification. The arguments are shown by constructing false headers to trick the MalConv model. By manipulating the informative bytes which constitutes less than one percent of the malware content, they show that more than fifty percent of the altered malware samples are classified as false-negative by MalConv. Suciu et al. [31] investigate the sensitivity of MalConv architecture by designing adversarial attacks to the architecture, and claims that it does not encode positional information of input features well, and vulnerable to append based attacks. Song et al. [32] introduce a reinforcement learning guided framework to design adversarial samples and attack MalConv and EMBER. We take these observations into account in our experiments.

Lu et al. [33] propose a simple and shallow self-attention transformer model that focuses on low-latency classification task. They try modeling the malware input both as a 1D sequence and as a 2D image to train their model. An extensive benchmark is performed by using the models such as MalConv, CNN-BiLSTM, CNN-BiGRU. Amin et al. [34] benchmark various deep learning architectures such as CNN, DAE, DBN, RNN, LSTM and BiLSTM, and reports their performance on Android malware classification, including zero-day benchmarks. For readers interested in an extensive summary of malware classification modeling with learning based approaches, we direct the reader to [35] where many recent approaches are addressed in a taxonomic manner.

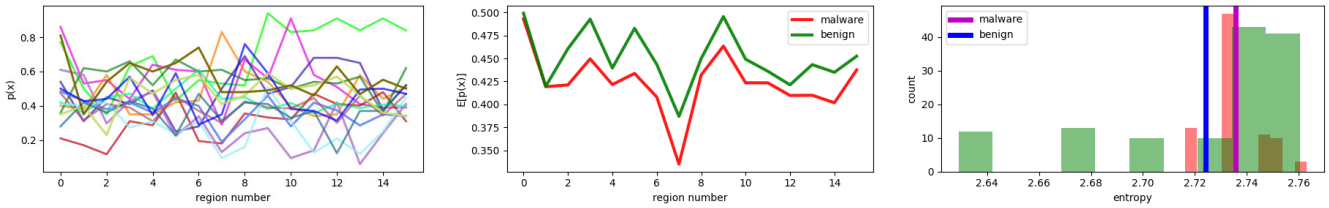
III. PROPOSED APPROACH

A. LOCAL ANALYSIS DIFFICULTY OF MALWARE

Most of the works in the literature treat malware detection as a classification problem. Some of these work report sufficiently high performances on their datasets, and their performance appears to be mostly saturated on many datasets. This raises the question whether malware detection is a solved problem or if the datasets used lack representativeness to highlight the difficult of the task. If the complexity of the problem indeed low, it is natural to try to understand classifiers' responses as in Riber et al.'s work [36]. If this hypothesis is true, a demand for of interpretability should be sought out. In this sense, we believe any analysis that could shift the domain towards explainable AI [37] would benefit the malware analyst.

To enhance explainability, it is interesting to understand whether certain parts of a malware file is more informative than the others. If some local parts are more interesting than others, then training a classifier over subsets of malware could not only yield increased interpretability but also hint towards localizing the regions that characterize the malware-ship. However, without loss of generality, it is quite difficult and expensive for most malware researchers to obtain a comprehensive line of code analysis providing which local regions are directly attributed to being a malware. Due to this limitation, we instead test the locality hypothesis by creating an alternative dataset where equally divided sections of malware and benign samples are used to form training samples for the classifier.

In the first step, we learn opcode vector features in an unsupervised fashion [38]. This is followed by equally partitioning local regions for each malware file, and representing each x_r region with its average opcode vector. Then we train a simple region classifier using averaged opcode features. Then we evaluate the posterior probability $p(y = 1|x_r)$ of the classifier for each local region. In Figure 1a, we show these posterior probabilities $p(y = 1|x_r)$ for different samples. Although we do not fully expect an ideal unimodal profile with few peaks, we would at least expect some clues for discrimination. At least for benign samples, a consistent and concentrated signal in these posterior probabilities would be expected where we know all regions should output a very small posterior probability of being malware. The analysis did not provide us this discrimination. Moreover, in Figure 1b, we plot the expected value of posteriors for each local region $E[p(y = 1|x_r)]$ of malware and benign samples. Two signals exhibit very similar characteristics, hinting that it is quite



(a) Posterior $p(y = 1|x_r)$ in each region. (b) Average posterior probability per region. (c) Entropy distributions of posteriors.

FIGURE 1. a) Posterior probabilities $p(y = 1|x_r)$ of classifier for each region of the samples after training. The responses are highly dispersed, it is quite difficult to observe an uniform, and peaky pattern which attributes the predictions to a single region. b) Average posterior probability per region for both malware and benign sets. Two signals exhibit very similar characteristics. c) Histogram of the entropies of posterior probabilities, malware (red) and benigns (green). Distinguishing signals are even difficult with the entropy information.

difficult to capture any clue of interpretability from classifier responses.

Lastly, motivated by the significant role of entropy and its role in the modeling of disorder in words and word vectors [39], [40], we quantify the amount of disorder in these posterior probabilities. For a malware sample we compute its signal entropy $H_m(.)$:

$$H_m[p(y = 1|x)] = -\sum_{x_r} p(y = 1|x_r) log\,(p(y = 1|x_r))$$

for both malware and benign sets. Figure 1c shows the entropy histogram for both s $H_m[p(y = 1|x)]$ (light red) and $H_b[p(y = 1|x)]$ (green). Although benign samples' entropy has a higher standard deviation, as expected, both sets have a very close average entropy (showed by vertical lines) hinting that the amount of disorder of posterior probabilities in our local analysis are almost the same for both sets. This analysis suggests that achieving interpretability for which code regions are responsible for malware-ship is very difficult, and we require a convolutional approach which learns the malware as a single entity.

B. JOINT CONVOLUTIONAL MODELING

Based on these findings, we understand that it is nontrivial to train a localized classifier for addressing malware classification problem. Any label noise introduced from the non-malware regions immediately hurts the classifier. Although this observation lays out a difficulty for moving towards explainable solutions, they are on par with the findings of Raff et al.'s work [16], who suggests that any analysis can't be carried out properly if we segregate the holistic structure of the malware file. It is difficult to attribute meaning to particular individual subsets of the malware, and the complexity of this learning problem is higher than initially guessed by current works. Thus, additional mechanisms to properly address this complexity is required to keep the performance of the learner in control. This is especially important if the training is done under scarce data regime. To address this, we adopt a network which is based on a set of convolutional filters. Such convolutional solutions for malware classification task in literature is vast. MalConv [16] is one of these architectures, adopting a gated and nongated convolutional block to filter the raw byte of Windows malware. Our architecture exhibits similarities with MalConv but different to it in few aspects:

1) OPCODE BASED TRAINING

Many convolutional architectures including MalConv architecture is designed to operate on raw bytes of the malware file. Although this approach aims to retain all relevant information, it requires a huge amount of training data to be at disposal of the malware analyst. Full bytes include vast amount of irrelevant content and as the dataset size and the variety of malware increase, training using the raw-byte approach will introduce inefficiency. Findings of Kadri et al. [7] suggest that performances obtained from raw-byte MalConv training can be unstable if the training is not properly performed. In our work, given that the training data is scarce, we also believe that it is reasonable to learn these convolutional filters with the under the guidance of previously extracted opcode sequences rather than raw bytes. By reducing the input space's dimensionality, opcodes introduce a natural regularization to the convolutional filters and significantly reduces the amount of training time and required number of samples.

Another advantage of using pretrained opcode representations is the speed. In contrast to convolutional models with an initial embedding layer, we do not jointly learn such an embedding layer. Because opcode features are already pretrained, we freeze the embedding layer and focus on learning the convolutional filter weights. This tweak enables our model to learn an order of magnitude faster. A Second reason for using opcodes is that there have been discussions on MalConv architecture's susceptibility for attacks since it usually exploits information from the file header [29]. Thus, we expect the opcode based training to circumvent this weakness and mostly focus on instructions to capture a malware-ship pattern.

2) TARGET REGULATED CLASSIFICATION

Our second strategy is to introduce weak target labeling procedure to the learning problem. It is known that each malware file originates from a specific family, and each family exhibits predefined and limited behaviors, denoted

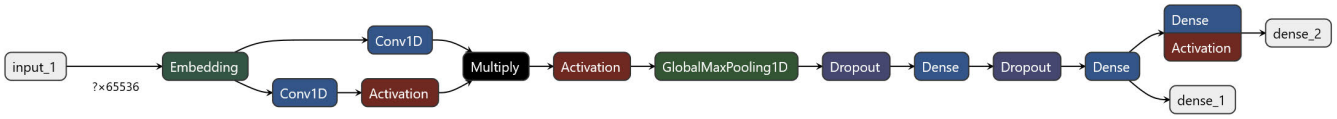


FIGURE 2. Overview of our model architecture.

as $\mathbf{z} = \{z_1, z_2, z_3\}$. For example, some malware might perform a *renaming* operation whereas a sample from the same family can perform *string encryption*. Our key idea here is to first incorporate these task information as attributes $\mathbf{z}$, along with their dependencies, and then cast the training as a multi-target to have a more accurate malware classification. Thus, in this step the convolutional architecture learns the relationship $f_\theta(\mathbf{z}|x)$ where x is a training sample and θ is the set of parameters and $\mathbf{z}$ are the known set of behavioral target variables.[1] The final architecture is shown in Figure 2.

To identify these dependencies, we analyzed the correlations between traits of the Android malware families. Out of 70 families, we selected the most frequent malware, and run a correlation test over their traits. We observed both positive and negative correlations between the types of tasks these malware perform. For instance, it is statistically likely for a random malware to do *string encryption* if it performs *dynamic-loading*, while it is less likely for it to have *native payload* if it uses a *renaming* approach. These patterns motivate us that using these dependencies effectively can help to train a neural architecture better. For this purpose, we resort to multi-target classification formulation to effectively incorporate these task dependencies into the problem.

We anticipate that such a multi-target formulation will introduce an auto-regularization effect our malware classification model. We acknowledge that it is quite likely to learn a pattern that correlates well with malware characteristics but does not necessarily indicate the presence of malware. However, it's important to note that selected malware from particular family does not necessarily possess all malicious tags associated with the family, but is likely to have some of them in its code base. Thus, it is more logical to think of these attributes as the so called weak labels with lower reliability, preparing the architecture to correctly predict malware/benign decision.

C. FINAL FORMULATION

In the most general scenario, some behavioral attributes of these malware samples can have nonlinear dependencies. However, we stay agnostic and do not assume any nonlinear relation between these attributes. Instead we expect the network to learn these priors through the data. Following this direction, we avoid any weighting of the traits during the learning. When some malware samples in our dataset have no tagged attribute, we treat these samples pessimistically, and assume that the respective malware family is likely to possess all of the attributes. Thus, these samples are treated to have an attribute vector full of ones $\mathbf{1}_K$.

As our convolutional network will output positiveness of each trait, this information has to be somehow linked to the final malware or benign decision. In here, we have conducted previous experiments to translate the multi-target regression output to final binary decision. Initial strategies of vector thresholding $\frac{1}{K}\sum_{i=1}^{K} p(z_i|X) > \frac{1}{K}$ where K is the number of targets used, did not suffice to discriminate the malware-ship.

Thus, here we incorporate another dense layer to learn the relation of weak target predictions to the final malware $g_\phi(y|\mathbf{z})$ where y is the final prediction and $\mathbf{z}$ is provided by our convolutional architecture as in the form of $\hat{\mathbf{z}}$. Since there is no sequential pattern between $\hat{\mathbf{z}}$ and y, a dense layer would suffice to capture the nonlinear relation between these variables. The pipeline is trained end to end fashion to avoid any information loss that might be introduced due to the separated training. Thus for a given sample x, the objective function of our model becomes the linear combination of two different loss functions:

$$J(\theta) = \lambda \ell_M(\mathbf{z}, f_\theta(x)) + (1-\lambda)\ell_C(y, g_\phi(f_\theta(x))) \quad (1)$$

where θ, ϕ are model parameters and ℓ_M is the Mean Squared Error (MSE) and ℓ_C is the Cross Entropy loss function. λ the parameter that weights learning of $f(.)$ and $g(.)$ functions.

IV. EXPERIMENTAL RESULTS

Experimental Setup: The experiments are conducted on a workstation PC having 128GB of RAM and Ryzen 9 5900 CPUs. Unless otherwise stated, all experiments are run on RTX A5000 with 24GB VRAM with 8192 Cuda Cores. For the software, we use Tensorflow version 2.12 and Cuda version 11.8. We set batch sizes of 16 in order to have a balanced trade off between experimental speed and accuracy. No batch normalization is done as it is known to degrade performances.

Experimental Evaluation: We mainly evaluate the performance of the models using their training, test and zero-day accuracies. Moreover, in order to understand the robustness of the models, we test model performances on various sensitivity analysis settings. In order to leave out randomness in the GPU calculations and ensure experimental reproducibility, we repeat each experiment with five different seeds and report the mean accuracy.

Tested Models: We use the following set of models both on Android and Windows experiments:

[1] List of variables are provided in the Supplementary Material.

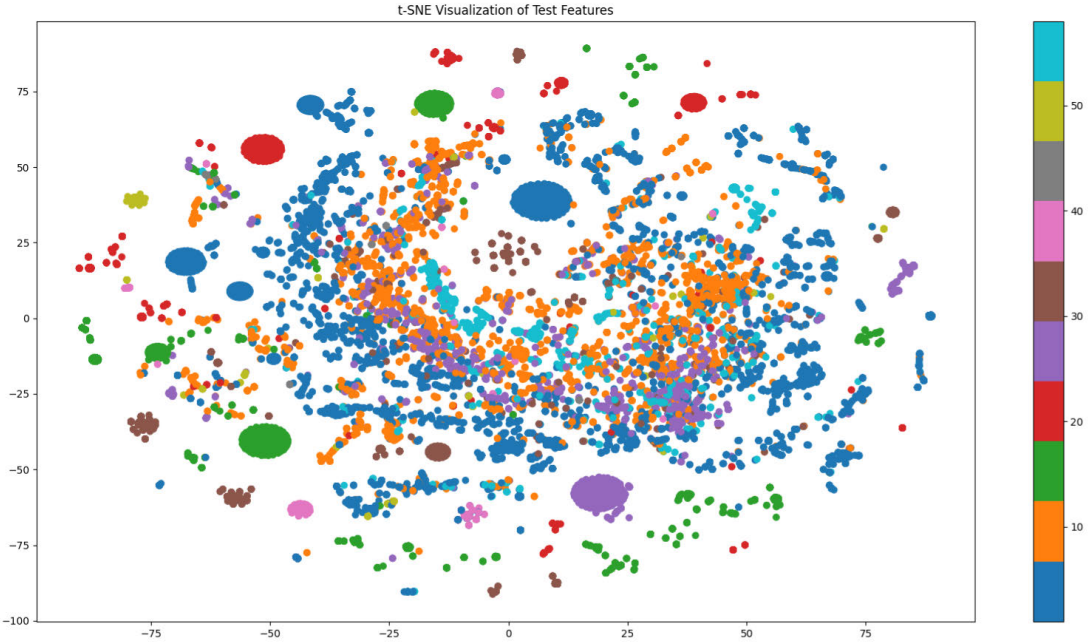


FIGURE 3. T-SNE embeddings of average vector representations of malware files, color coded by their respective family. Best viewed in color.

TABLE 1. Number of samples for each family in the AMDArgus dataset.

Airpush	7843	FakeAngry	10	GoldDream	53	Mecor	1820	Roop	48	Ztorg	17
AndroRAT	46	FakeAv	5	Gorpo	32	Minimob	203	RuMMS	402	Triada	210
Andup	43	FakeDoc	21	Gumen	145	Mmarketpay	14	SimpleLocker	165	Univert	10
Aples	21	FakeInst	2168	Jisut	548	MobileTX	17	SlemBunk	174	UpdtKiller	24
BankBot	648	FakePlayer	21	Kemoge	15	Mseg	235	SmsKey	165	Utchi	12
Bankun	70	FakeTimer	12	Koler	69	Mtk	67	SmsZombie	9	Vidro	23
Boqx	215	FakeUpdates	5	Ksapp	36	Nandrobox	76	Spambot	15	VikingHorde	7
Boxer	44	Finspy	9	Kuguo	1198	Obad	9	SpyBubble	10	Vmvol	13
Cova	17	Fjcon	16	Kyview	174	Ogel	6	Stealer	25	Winge	19
Dowgin	3385	Fobus	4	Leech	128	Opfake	10	Steek	12	Youmi	1301
DroidKungFu	546	Fusob	1276	Lnk	5	Penetho	18	Svpeng	13	Zitmo	24
Erop	46	GingerMaster	128	Lotoor	328	Ramnit	8	Tesbo	5	Total	24516

- APK2Vec [38]: A Skip-Gram [41] based opcode vector averaging algorithm to perform malware classification. Under some conditions, this model is shown to yield satisfactory testing accuracies on the AMDArgus malware dataset. For final classification with APK2Vec's average vectors, Random Forest Classifier used.
- CNN-BiLSTM [25]: A well suited LSTM originated architecture with a CNN backend, proven to have high performance on many tasks due to initial CNN filtering and BiLSTM extension.
- MalConv [16]: A gated CNN model that outperforms many architectures on malware classification tasks.
- TRConv. Our Target Regulation Convolution model with additional parameter λ.

A. ANDROID APK EXPERIMENTS

Dataset Collection and Preprocessing: We form our dataset from a collection of sources. We obtain most of the malware samples from the AMDArgus dataset. This dataset have approximately 24k Android applications. The timespan of the Android applications in this dataset lies between 2012 and 2016. In order to obtain a well balanced benign set of samples, we aggregate samples from MalAnal2017 [42], Maldroid2020 [43] and also some extra benign samples manually collected from multiple sources. In total, this aggregation strategy outputs 22899 malware, 22279 being samples. APK malware are originated from a total of 71+1 families. Table 1 shows the malware family distribution. Families such as Airpush, Dowgin and FakeInst have significantly more samples than other families.

We used open source ApkTool software for Android to extract the opcode sequences from APK files. During extraction, various non-essential files such as resources and assets within the APK are excluded. Extraction process generates multiple *.smali* files for single APK, containing human-readable assembly language instructions specific to the Android application environment. These resulting

.smali files are then concatenated into single *.smali* file. To combat overfitting in the context of limited dataset availability, we strip all parameters, operands, and addresses from the instruction sequences during opcode extraction, retaining only the opcodes themselves. Otherwise it is quite likely for models to overfit based on the values of these parameters. Additionally, to standardize the sequence length and accommodate variations in file sizes, we set a default window of the first 65k opcodes for each file. For APKs with fewer than 65k opcodes, zero-padding is applied to extend the sequences, while longer sequences are truncated to fit within the 65k opcode window.

In Figure 3, we visualize t-Distributed Stochastic Neighbor Embedding (T-SNE) plot of the malware samples where each color represents a malware family. Families having high number of samples are mostly dispersed in the feature space. Observe how some malware families form locally concentrated clusters. We believe one of the reasons for this phenomenon is that the dataset contains custom clones of some particular representative malware samples.

In malware analysis literature, zero-day analysis is an indicator of well-generalization. The point in this analysis is to deliberately leave out samples from some of the families during the training, and only train on the samples coming from the rest of the families. Then, it is expected from the classifier to generalize well and obtain a good prediction performance on samples from unseen families.

We explore zero-day performance of our model and try to infer its robustness against zero-day malware. Previously [44] performed a zero-day analysis for the largest 20 family, and left out of them in turn from the training. However, we believe that such a zero-day test is slightly skewed. It is also optimistic for assuring a well real time performance because the effect of class-imbalance is completely left out from the problem by selecting zero-day samples from the most frequent families. Thus, we follow a more balanced and heterogeneous leave-out strategy, and randomly choose families that contain sufficiently large number of samples. Some work consider allowing one exemplar sample in the training set. Here, as we aim to operate on a slightly pessimistic scenario, we do not include not even one sample from these families to the training set. In this sense, our zero-day experimental design can be interpreted as the worst case scenario.

Parameters/Hyperparameters: For the CNN-BiLSTM architecture, we performed hyperparameter search and used the following parameters: a batch size of 16, 10 max epochs, early stopping by validation set as the stopping criterion, dropout: 0.2, 128 LSTM units, CNN kernel size of 32 with a stride 32 and pooling of 2. 128 CNN filters and 64 dense layers neurons in total. 239 unique opcodes are present. This results in 417k total parameters for CNN-BiLSTM. (263k is from LSTM layer).

For APK2Vec training, we used 32 dimensional embeddings with window size of 3. As architecture, we choose the standard Skip-Gram architecture with Negative Sampling with $k = 10$ and frequency subsampling parameter $6e-5$.

For our model, The parameter λ governs how much our model prioritizes predicting the weak target variables and final malware-ship decision. We pessimistically treated all malware from a particular family to share their family's weak-target label. This assumption might not hold for some members of the family, thus we do not desire our model to exactly learn these capabilities. Thus, to determine this parameter we have relied on empirical observation and optimization landscape inspection, and fixed it to $\lambda = 0.2$ for all experiments. For optimization, we use RMSProp algorithm. This algorithm maintains a moving average of the square of gradients. For RMSProp, we use learning rate $1e-4$, old gradient discount factor $\rho = 0.9$, epsilon tolerance rate of $1e-07$ with no momentum applied. A dropout of 0.2 is applied for models. We use a batch size of 16. We apply random shuffling of the instances at every step of optimization.

TABLE 2. Test evaluation for android dataset.

Model	ACC	FP	FN	TP	TN	F1
MalConv	0.9551	313	295	6575	6371	0.9551
CNN-BiLSTM	0.8729	1246	476	6394	5438	0.8718
Apk2Vec	0.9435	384	382	6488	6300	0.9435
TRConv	0.9548	430	183	6687	6254	0.9548

TABLE 3. Zero-day evaluation for android dataset.

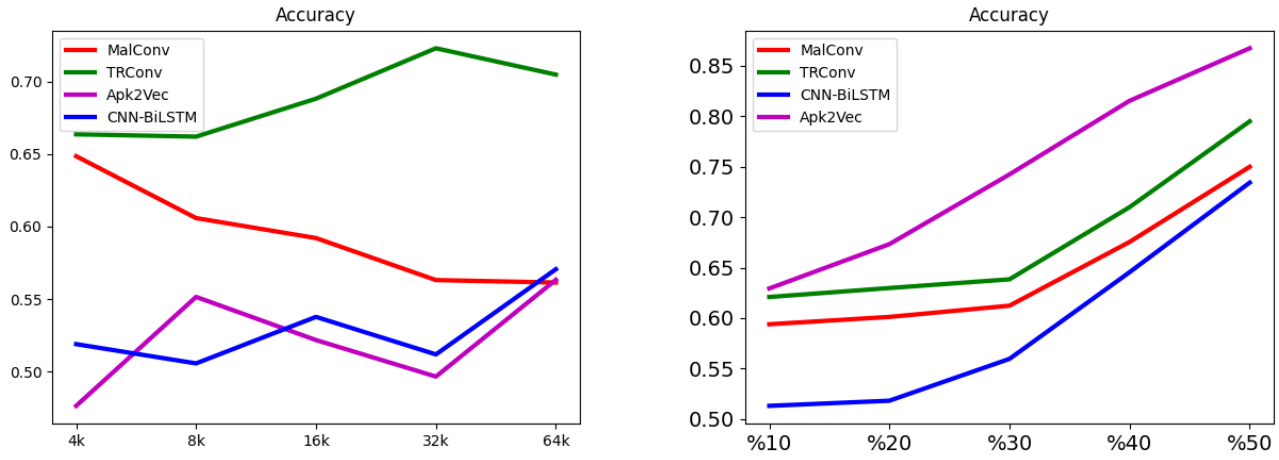
Model	ACC	FP	FN	TP	TN	F1
MalConv	0.5614	0	709	908	0	0.3531
CNN-BiLSTM	0.5706	0	694	923	0	0.3583
Apk2Vec	0.5633	0	706	911	0	0.3603
TRConv	0.7049	0	477	1140	0	0.4121

After our analysis of weak target variables, we choose 15 target variables. Full list of behavioral target variables for Android malware is shown in Table 8

1) RESULTS ON OPCODE LENGTH

Raff et al. [45] proposed a solution for classifying malware sequences of extreme length. However, for many architectures such as BiLSTMs, opcode length parameter is still a design criterion, and it is not clear how the length of the opcodes influence the final performance. In this experiment, we query the limitations of benchmarked models against variations to the representations in opcode sequence lengths.

Since the concatenation order of *.smali* files is randomized, we expect to observe a gradual linear increase in performance as the considered opcode sequence length increases. Prior work in [25] tested the opcode length parameter space from 2k to 10k. According to the statistics in our dataset, 38% of test set has more than 65k opcodes, and 40% of zero-day malware has more than 65k opcodes. Motivated by this significant proportion of large malware files, we extend the analysis to a larger parameter space to minimize the chances



(a) opcode length, zero day performance

(b) exclude benign, zero day performance

FIGURE 4. Malware classification on ARGUS dataset. a) performance for varying opcode lengths b) performance when some benign samples are excluded from the experiments x-axis is the percentage of benign samples excluded.

of discarding influential sections that could be crucial for classification.

In Figure 4a, we show performances on zero-day set for a regime of 4k to 64k opcodes. As opcode length increases, we observe that the performance of CNN-BiLSTM and APK2Vec architectures enjoy a minor gain. For APK2Vec, the performance appears to be greater on longer opcode sequences. We believe this is due to fact that APK2Vec is insensitive to the number of vectors it is averaging. The more opcode vectors are used, the better for it. Similarly for LSTM based approaches, we observe it is safer to train with the longest possible opcode length. MalConv especially suffers a steep performance drop. It is possible that when opcode length increases, MalConv network will probably undergo a label shift in which network filters will change what they are learning. Our model also undergoes a small performance degradation. This could be an indicator that, on the short term, it becomes harder to learn the traits simultaneously as input size increases. However, the performance degradation is not substantial, and our model is on the average still superior to the baselines.

2) RESULTS ON THE EFFECT OF CLASS IMBALANCE

Class imbalance is a common occurrence in malware classification tasks, particularly in multi-class scenarios. Allix et al. [46] argue that adding more benign samples to the training increases precision but decreases recall. Thus, they claim decreased malware/benign ratio will in general output worse performance. We have verified this by excluding [%10, %20, %30, %40, %50] benign samples from the dataset and found out that in general, average accuracy of models decrease as the imbalance is on benign samples' favor. This phenomenon can be explained by the entropic difference between malware and benign samples. As more benign samples dominate the training set, the average entropy of the training set will increase relatively, which in turn will make it harder to classify the malware without further precaution or treatment. Thus, we are inclined to believe that collecting malware samples are much more helpful for an effective learning in limited data regime.

Figure 4b shows our results over zero-day set. As a general trend, all models output better zero-day performance. As we exclude more benign samples from the training, models are more prone to label a zero-day sample as malware. Up to %20 exclusion, models output a minor performance gain whereas APK2Vec's decision rapidly changes. Although this performance increase can be interpreted as a gain for APK2Vec, we suspect that APK2Vec is too quick to embrace the class prior imbalance introduced in the benign-exclude settings. So from this point of view, we believe that minor performance gain of models is a sign that they are simply learning well, and not effected immediately by the inherent training data class imbalance up to %20 exclusion.

TABLE 4. Subtype information of the malware in the AMDArgus dataset.

Ransom	Trojan-Banker	Trojan-Dropper	Trojan-SMS
Trojan	Trojan-Spy	HackerTool	Trojan-Clicker

3) RESULTS ON ZERO-DAY ANALYSIS

In Figure 5 we plot the average test and zero-day accuracies of randomly trained from subset of dataset. At each step, a larger subset of training data is used to train models. As training data size increases, performance of APK2Vec and CNN-BiLSTM degrades which suggests that high performances obtained with these models might be due to overfitting. As training set size increases, our model is able to output a better zero-day performance which is an indicator that it is capable of generalizing. Highest performance is achieved when we use the full set. The risk of overfitting increases as samples in the training set is poorly sampled, or weighted from arbitrary sources. As the difficulties constructing

a representative dataset always remains, extra categorical information immediately helps to train the malware detector. We believe this trends of weaknesses will be similar to other architectures if training set is not properly maintained.

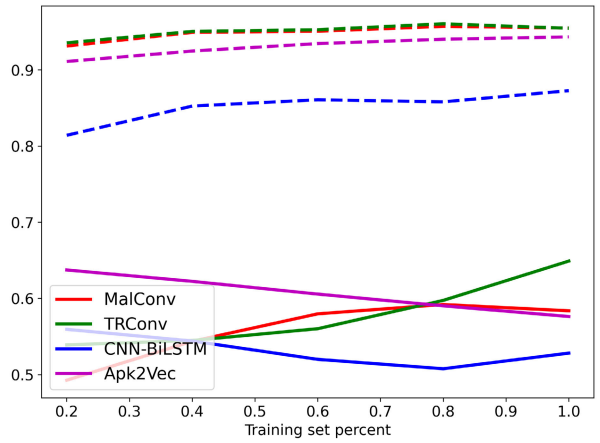


FIGURE 5. Test set (dashed curves) and zero-day (non-dashed curves) accuracy of models in AMDArgus dataset.

B. WINDOWS EXPERIMENTS

In this section, we aim to explore our model's performance on Windows Operating System Environment and show results on PE (Portable Executable) files.

1) DATASET AND PREPROCESSING

Following the recent advances on static malware analysis, we use the Malware Open-source Threat Intelligence Family (MOTIF) dataset [47]. This dataset is the first dataset to provide a comprehensive alias mapping that includes aliases derived from both open-source reporting and antivirus signatures. It contains a larger and more diverse set of malware samples. Malware samples in this dataset has low labeling inconsistency compared to other malware source dataset, yielding an appropriate source for training the models. The samples are originally collected from 329 families. After omitting families which have less than two samples, and performing the disassembling approach, we are left with 2601 samples from 295 families. Table 1 shows the histogram of the number of samples per class. For Windows PE disassembly, we used executable reversing tool PE-Bear [48]. We considered all programs sections (not only the .text section that primarily contains the code) because malware tend to hide codes into different named sections. We then extract opcodes from the resulting code.

As the MOTIF dataset has no benign samples, addressing malware vs benign task requires introduction of benign samples. To solve this need, we use recent PEMDB [49] dataset to enlarge the training set with an extra 2K benign samples. Similarly to the work in [38], we also add 100 malware samples from PEMDB dataset to the training set, and 1400 of them to the zero-day dataset. As most of the training and validation accuracies are uninformatively high, we directly report zero-day and test set accuracies to assess how models respond to mostly unseen malware samples.

We use 70/30 split to training and test set using a sampling strategy to ensure that each family has at least one sample in both sets. Opcodes are extracted in a similar manner to the APK experiments. However, we notice that opcodes exhibit higher amount of diversity in the PE experiments with respect to APK scenario. In the APK experiments we had 239 distinct opcodes, whereas in Windows experiments we observed 1536 distinct opcodes in our dataset. Our visual analysis suggests that most of the time, the groupings in APK opcode embeddings are easier to notice than the PE experiments which makes sense due to the scarce data and increased number of opcodes.

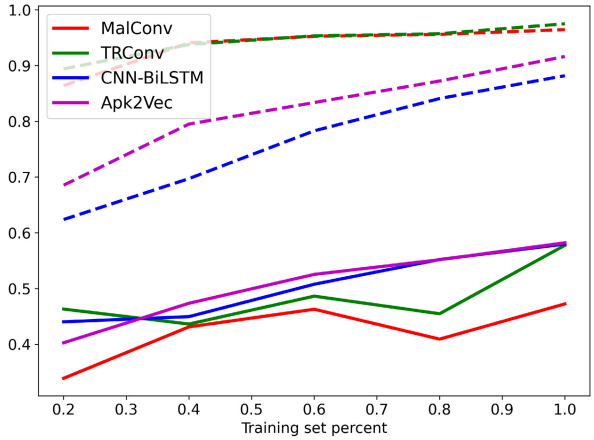


FIGURE 6. Test set (dashed curves) and zero-day (non-dashed curves) accuracy of models in the MOTIF dataset.

MOTIF dataset offers malware description on family level. However, as the native structure of PE files are much more complex than APK programs, this complexity makes learning architecture design harder. For this reason, a higher amount of information should be incorporated with weak target variables to be able to achieve representative features with a small amount of training set. To address this difficulty, we extract weak target variables using CAPA tool by Mandiant. This tool scans the PE file and extracts a list of hierarchical attributes that characterize the PE file. The full list of attributes are provided on Table 7. To maintain simplicity, we binarize this list of attributes, considering a higher-level attribute as present if any of its subattributes belong to it. Thus the counts of identity of low level attributes are discarded. This approach gives us 19 weak target variables. The preprocessing is automated using relevant libraries for both PEBear and the CAPA tool.

2) RESULTS AND DISCUSSION

The Windows experiment presents more challenges compared to the Android experiment for several reasons. Firstly, translating PE files into common instruction sequences is more difficult because some samples are for various runtime environments while others are native. Secondly,

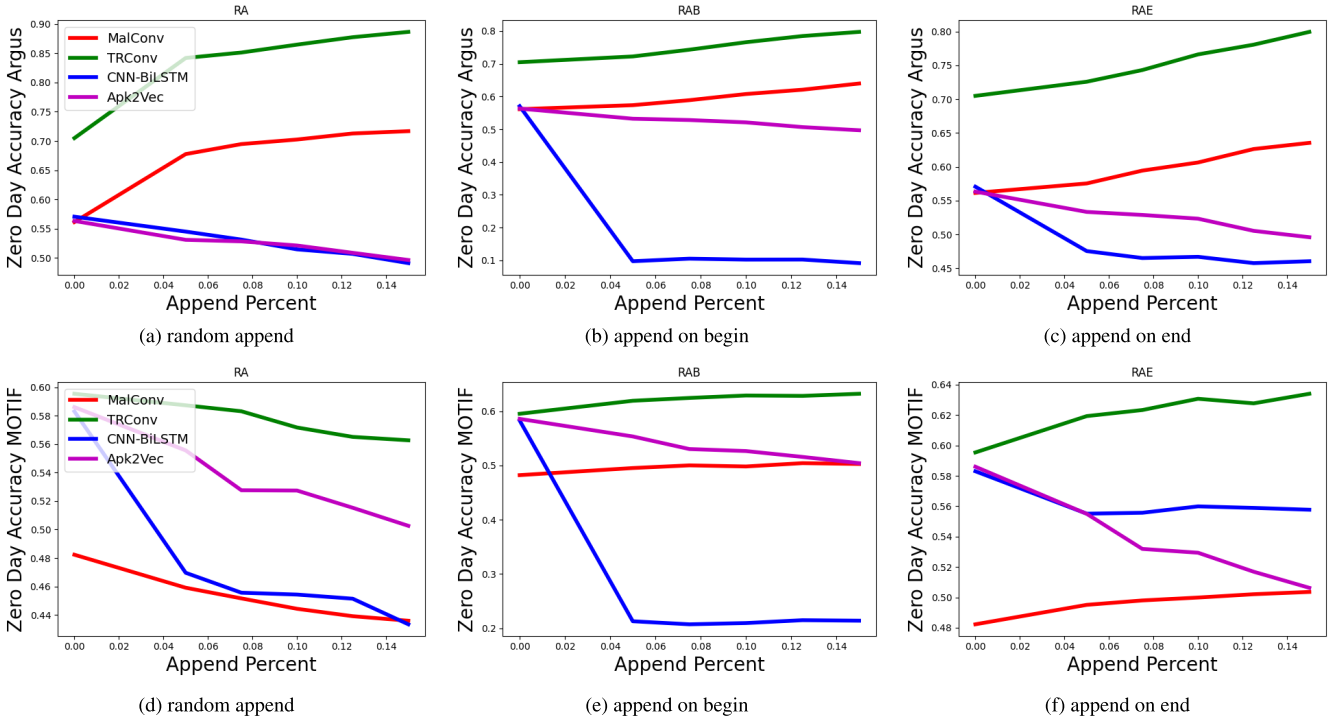


FIGURE 7. Sensitivity result analysis for each model on both (AMDArgus for APK and MOTIF for Windows) dataset. Top row: attacks of a) random append b) append on begin c) append on end on AMDArgus dataset. Bottom row: attacks of a) random append b) append on begin c) append on end on the MOTIF dataset.

while Android is entirely object-oriented, making it easy to find opcodes of classes, Windows architecture does not have a strict limitation to be object-oriented. Thirdly, compilers can obfuscate code which increase complexity. Moreover, in Portable Executable files, all code are not necessarily on .text section, some of the codes can be located in other segments which introduces extra difficulty to the preprocessing step.

TABLE 5. Test evaluation for windows dataset.

Model	ACC	FP	FN	TP	TN	F1
MalConv	0.9647	8	23	638	216	0.9550
CNN-BiLSTM	0.8820	13	92	569	211	0.8596
Apk2Vec	0.9164	2	72	589	222	0.8990
TRConv	0.9751	12	10	651	212	0.9668

TABLE 6. Zero-day evaluation for windows dataset.

Model	ACC	FP	FN	TP	TN	F1
MalConv	0.4823	0	725	675	0	0.3160
CNN-BiLSTM	0.5830	0	584	816	0	0.3672
Apk2Vec	0.5860	0	580	820	0	0.3693
TRConv	0.5954	0	566	834	0	0.3696

In Figure 6, we plot accuracies of each model. Here, at each step, models see a portion of the training data. Thus, in the low sample regime, models suffer from data scarcity. In Tables 2 and 3, we present accuracy and F1 scores for the Android dataset. Similarly, Tables 5 and 6 provide the corresponding metrics for the Windows dataset. In general, TRConv, CNN-BiLSTM, and APK2Vec appear to have a good performance on the zero-day set. On the test set, TRConv and MalConv perform the best. Also, the relatively lower zero-day accuracy can be attributed to the non-informativeness of the benign samples, which is difficult to form a representative subsample due to intellectual property. However, the experiment acts as a proof of concept for our model, which is well applicable to the Windows architecture with a minimal amount of changes. Our model maintains a relatively higher performance compare to MalConv due to learning with weak target variables.

C. SENSITIVITY ANALYSIS

1) EXPERIMENTAL SETUP

In this section we analyse how each model responds to random injections to input opcode sequences. After training the models, we change the opcode representation of test and zero-day samples by applying random opcode injections in order to evade detection. Most of the work focusing on designing attacks implements them as injection of random or previously designed bytes as in [30]. In this work, we focus on opcode level injections. We differentiate between the following attack strategies:

- RA: scenario that inserts random opcodes at random locations of the opcode sequence. This strategy completely breaks the opcode sequence pattern in the file, and can be thought as the most difficult type of attack.

- RAB: inserting opcodes to the beginning of the opcode sequence. Some convolutional architectures such as MalConv is known to quickly scan the file header which usually resides in the beginning of the file, this type of attack can be conceptualized as a measure of positional invariance whether the candidate model explicitly prioritizes the patterns in the beginning of the file.
- RAE: As a last strategy, we inject opcodes to the end of the opcode sequence. The rationale here is to check whether redundant end bytes (such as program sections where resource files exist) can deceive the convolutional filters.

In all experiments, we start with 5% injections and gradually increase the injection amount to 15% of the opcode sequence length. To keep the attacks simple, we sample attack opcodes from an uniform distribution. More complex attack techniques are possible by taking co-occurence statistics of opcodes into account. However, this is beyond the scope of this paper and we leave that as future work.

2) QUANTITATIVE RESULTS

We plot our findings on Figure 7. In the AMDArgus dataset, MalConv and TRConv appears to be robust to the variations in the test set and zero-day set. As opcode injection proportion parameter increases, these two models classify more zero-day files as malware. This might be due to the fact that it is harder for MalConv and TRConv to overfit since the model complexity is relatively lower with respect to the CNN-BiLSTM based architectures. APK2Vec suffers from a small performance drop. CNN-BiLSTM undergoes sharp performance drops. It totally fails to predict zero-day when RAB attacks are performed. The memory neurons of CNN-BiLSTM gets confused very quickly.

In Figure 7b, we show attack results for the MOTIF dataset. Under RA attacks, performance of all models degrade quickly. Having less training data appears to hurt all models immediately. In RAE attacks, APK2Vec significantly loses most of its prediction power while for rest of the models a minor performance change is observed.

In general, we observe that MalConv and TRConv suffers more from random attacks in the MOTIF dataset compared to AMDArgus dataset. Morever, on both scenarios, LSTM model is effected most when we randomly add opcodes to the beginning of the malware. This observation highlights that any LSTM training methodology should by default augment the training set with shifted opcode augmentations. However, while designing this type of augmented training is easy for image-based input without breaking the integrity of the image. For opcode sequences it is not straightforward to artificially create valid opcode sequences. Thus with TRConv we can still obtain reasonable performance without using such additional augmentation mechanism.

TABLE 7. List of used weak target variables for windows architecture.

Windows Selected Capabilities	Token Value
DEFENSE EVASION	1
DISCOVERY	2
EXECUTION	3
PRIVILEGE ESCALATION	4
ANTI-BEHAVIORAL ANALYSIS	5
CRYPTOGRAPHY	6
DATA	7
FILE SYSTEM	8
OPERATING SYSTEM	9
PROCESS	10
COLLECTION	11
IMPACT	12
PERSISTENCE	13
ANTI-STATIC ANALYSIS	14
COMMUNICATION	15
MEMORY	16
COMMAND AND CONTROL	17
CREDENTIAL ACCESS	18
HARDWARE	19

TABLE 8. List of used weak target variables for android architecture.

Android Selected Capabilities	Token Value
RENAMING	1
STRİNG_ENCRYPTİON	2
DYNAMIC_LOADİNG	3
NATIVE_PAYLOAD	4
EVADE_DYNAMİC_ANALYSİS	5
ADWARE	6
BACKDOOR	7
RANSOM	8
TROJAN-BANKER	9
TROJAN-DROPPER	10
TROJAN-SMS	11
TROJAN	12
TROJAN-SPY	13
HACKERTOOL	14
TROJAN-CLİCKER	15

V. CONCLUSION AND DISCUSSION

We proposed a novel opcode based malware classification approach where weak target regulation variables are constructed and used inside a convolutional framework. The performance of the proposed method has been extensively evaluated on a zero-day analysis for both Android and Windows platform malware. Its robustness is evaluated on sensitivity analysis tasks. It is quite helpful to jointly learn these weak target regulation variables to classify malware.

We assumed all weak target variables are independent and there is no complicated relation between the target variables.

As a future work, it would be interesting to model relations of these variables as in [50] and incorporate that extra information to the network to obtain a more effective training. Another future direction could be measuring the limitations of TRConv on more carefully designed attack strategies, i.e. smarter opcode insertion attacks that preserve program integrity. Finally, it would be interesting to explore whether we can transfer knowledge between different platforms to detect malware patterns faster.

APPENDIX

See Tables 7 and 8.

ACKNOWLEDGMENT

The authors appreciate malware analyst ''hasherezade'' for enhancements to her PEBear tool, which greatly facilitated the disassembly of Windows samples for this research.

REFERENCES

[1] J. Jiang, S. Li, M. Yu, G. Li, C. Liu, K. Chen, H. Liu, and W. Huang, "Android malware family classification based on sensitive opcode sequence," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jun. 2019, pp. 1–7, doi: 10.1109/ISCC47284.2019.8969656.

[2] C. Catal, H. Gunduz, and A. Ozcan, "Malware detection based on graph attention networks for intelligent transportation systems," *Electronics*, vol. 10, no. 20, p. 2534, 2021, doi: 10.3390/electronics10202534.

[3] K. Aktas and S. Sen, "UpDroid: Updated Android malware and its familial classification," in *Secure IT System* (Lecture Notes in Computer Science), vol. 11252, Nov. 2018, pp 352–368.

[4] J. Yan, Y. Qi, and Q. Rao, "LSTM-based hierarchical denoising network for Android malware detection," *Secur. Commun. Netw.*, vol. 2018, pp. 1–8, Nov. 2018, doi: 10.1155/2018/5249190.

[5] H. S. Anderson and P. Roth, "EMBER: An open dataset for training static PE malware machine learning models," 2018, *arXiv:1804.04637*.

[6] A. D. Raju and K. Wang, "LockBoost: Detecting malware binaries by locking false alarms," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Jul. 2022, pp. 1–8, doi: 10.1109/IJCNN55064.2022.9892667.

[7] M. A. Kadri, M. Nassar, and H. Safa, "Transfer learning for malware multi-classification," in *Proc. 23rd Int. Database Appl. Eng. Symp.*, 2019, pp. 1–7.

[8] R. Vinayakumar, M. Alazab, K. P. Soman, P. Poornachandran, and S. Venkatraman, "Robust intelligent malware detection using deep learning," *IEEE Access*, vol. 7, pp. 46717–46738, 2019.

[9] A. Pektaş and T. Acarman, "Learning to detect Android malware via opcode sequences," *Neurocomputing*, vol. 396, pp. 599–608, Jul. 2020, doi: 10.1016/j.neucom.2018.09.102.

[10] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *Proc. 10th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Oct. 2015, pp. 11–20.

[11] E. Raff, R. Zak, R. Cox, J. Sylvester, P. Yacci, R. Ward, A. Tracy, M. McLean, and C. K. Nicholas, "An investigation of byte n-gram features for malware classification," *J. Comput. Virol. Hacking Techn.*, vol. 14, pp. 1–20, Feb. 2018.

[12] Z. Ghezelbigloo and M. VafaeiJahan, "Role-opcode vs. opcode: The new method in computer malware detection," in *Proc. Int. Congr. Technol., Commun. Knowl. (ICTCK)*, Nov. 2014, pp. 1–6.

[13] N. Zhang, J. Xue, Y. Ma, R. Zhang, T. Liang, and Y. an Tan, "Hybrid sequence-based Android malware detection using natural language processing," *Int. J. Intell. Syst.*, vol. 36, no. 10, pp. 5770–5784, Jul. 2021, doi: 10.1002/int.22529.

[14] S. Yesir and I. Soğukpinar, "Malware detection and classification using fastText and BERT," in *Proc. 9th Int. Symp. Digit. Forensics Secur. (ISDFS)*, Jun. 2021, pp. 1–6.

[15] K. Lucas, M. Sharif, L. Bauer, M. K. Reiter, and S. Shintre, "Malware makeover: Breaking ml-based static analysis by modifying executable bytes," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2021, pp. 744–758.

[16] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole exe," in *Proc. Workshops 32nd AAAI Conf. Artif. Intell.*, 2018, pp. 268–276.

[17] O. J. Falana, A. S. Sodiya, S. A. Onashoga, and B. S. Badmus, "Mal-detect: An intelligent visualization approach for malware detection," *J. King Saud Univ.-Computer Inf. Sci.*, vol. 34, no. 5, pp. 1968–1983, 2022.

[18] J. Singh, D. Thakur, F. Ali, T. Gera, and K. S. Kwak, "Deep feature extraction and classification of Android malware images," *Sensors*, vol. 20, no. 24, p. 7013, 2020.

[19] M. Nisa, J. H. Shah, S. Kanwal, M. Raza, M. A. Khan, R. Damaseviciius, and T. Blazauskas, "Hybrid malware classification method using segmentation-based fractal texture analysis and deep convolution neural network features," *Appl. Sci.*, vol. 10, no. 14, p. 4966, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/14/4966

[20] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, "Novel feature extraction, selection and fusion for effective malware family classification," 2015, *arXiv:1511.04317*.

[21] J. Xu, W. Fu, H. Bu, Z. Wang, and L. Ying, "SeqNet: An efficient neural network for automatic malware detection," 2022, *arXiv:2205.03850*.

[22] A. Bensaoud, N. Abudawaood, and J. Kalita, "Classifying malware images with convolutional neural network models," *Int. J. Netw. Secur.*, vol. 22, no. 6, pp. 1022–1031, 2020.

[23] E. K. Kabanga and C. H. Kim, "Malware images classification using convolutional neural network," *J. Comput. Commun.*, vol. 6, no. 1, pp. 153–158, 2017.

[24] X. Xing, X. Jin, H. Elahi, H. Jiang, and G. Wang, "A malware detection approach using autoencoder in deep learning," *IEEE Access*, vol. 10, pp. 25696–25706, 2022, doi: 10.1109/ACCESS.2022.3155695.

[25] D. Dang, F. Di Troia, and M. Stamp, "Malware classification using long short-term memory models," 2021, *arXiv:2103.02746*.

[26] P. G. Balikcioglu, M. Sirlanci, O. A. Kucuk, B. Ulukapi, R. K. Turkmen, and C. Acarturk, "Malicious code detection in Android: The role of sequence characteristics and disassembling methods," *Int. J. Inf. Secur.*, vol. 22, pp. 107–118, Nov. 2022, doi: 10.1007/s10207-022-00626-2.

[27] D. Demirci, N. sahin, M. sirlancis, and C. Acarturk, "Static malware detection using stacked BiLSTM and GPT-2," *IEEE Access*, vol. 10, pp. 58488–58502, 2022, doi: 10.1109/ACCESS.2022.3179384.

[28] Y. Ding, S. Wang, J. Xing, X. Zhang, Z. Qi, G. Fu, Q. Qiang, H. Sun, and J. Zhang, "Malware classification on imbalanced data through self-attention," in *Proc. IEEE 19th Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom)*, Dec. 2020, pp. 154–161.

[29] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Explaining vulnerabilities of deep learning to adversarial malware binaries," 2019, *arXiv:1901.03583*.

[30] B. Kolosnjaji, A. Demontis, B. Biggio, D. Maiorca, G. Giacinto, C. Eckert, and F. Roli, "Adversarial malware binaries: Evading deep learning for malware detection in executables," in *Proc. 26th Eur. Signal Process. Conf. (EUSIPCO)*, Sep. 2018, pp. 533–537.

[31] O. Suciu, S. E. Coull, and J. Johns, "Exploring adversarial examples in malware detection," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2019, pp. 8–14, doi: 10.1109/SPW.2019.00015.

[32] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin, "MAB-malware: A reinforcement learning framework for attacking static malware classifiers," 2020, *arXiv:2003.03100*.

[33] Q. Lu, H. Zhang, H. Kinawi, and D. Niu, "Self-attentive models for real-time malware classification," *IEEE Access*, vol. 10, pp. 95970–95985, 2022, doi: 10.1109/ACCESS.2022.3202952.

[34] M. Amin, T. A. Tanveer, M. Tehseen, M. Khan, F. A. Khan, and S. Anwar, "Static malware detection and attribution in Android byte-code through an end-to-end deep system," *Future Gener. Comput. Syst.*, vol. 102, pp. 112–126, Jan. 2020.

[35] D. Gibert, C. Mateu, and J. Planes, "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges," *J. Netw. Comput. Appl.*, vol. 153, Mar. 2020, Art. no. 102526.

[36] M. T. Ribeiro, S. Singh, and C. Guestrin, "'Why should I trust you?'" *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 97–101, doi: 10.1145/2939672.2939778.

[37] P. Gohel, P. Singh, and M. Mohanty, "Explainable AI: Current status and future directions," 2021, *arXiv:2107.07045*.

[38] A. Egitmen, I. Bulut, R. C. Aygun, A. B. Gunduz, O. Seyrekbasan, and A. G. Yavuz, "Combat mobile evasive malware via skip-gram-based malware detection," *Secur. Commun. Netw.*, vol. 2020, pp. 1–10, Apr. 2020.
[39] C. Bentz, D. Alikaniotis, M. Cysouw, and R. Ferrer-I Cancho, "The entropy of words-learnability and expressivity across more than 1000 languages," *Entropy*, vol. 19, no. 6, 2017. [Online]. Available: https://www.mdpi.com/1099-4300/19/6/275
[40] T. Kekeç, D. Mimno, and D. M. Tax, "Boosted negative sampling by quadratically constrained entropy maximization," *Pattern Recognit. Lett.*, vol. 125, pp. 310–317, Jul. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167865519301424
[41] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. 1st Int. Conf. Learn. Represent. (ICLR)*Y. Bengio and Y. LeCun, Eds. Scottsdale, AZ, USA, May 2013, pp. 1–12.
[42] A. H. Lashkari, A. F. A. Kadir, L. Taheri, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark Android malware datasets and classification," in *Proc. Int. Carnahan Conf. Secur. Technol. (ICCST)*, Oct. 2018, pp. 1–7.
[43] S. Mahdavifar, A. F. Abdul Kadir, R. Fatemi, D. Alhadidi, and A. A. Ghorbani, "Dynamic Android malware category classification using semi-supervised deep learning," in *Proc. IEEE Int. Conf. Dependable, Autonomic Secure Comput., Int. Conf. Pervasive Intell. Comput., Int. Conf. Cloud Big Data Comput., Int. Conf. Cyber Sci. Technol. Congr. (DASC/PiCom/CBDCom/CyberSciTech)*, Aug. 2020, pp. 515–522.
[44] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 1–15.
[45] E. Raff, W. Fleshman, R. Zak, H. S. Anderson, B. Filar, and M. McLean, "Classifying sequences of extreme length with constant memory applied to malware detection," 2020, *arXiv:2012.09390*.
[46] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. Le Traon, "Empirical assessment of machine learning-based malware detectors for Android," *Empirical Softw. Eng.*, vol. 21, no. 1, pp. 183–211, 2016.
[47] R. J. Joyce, D. Amlani, C. Nicholas, and E. Raff, "MOTIF: A malware reference dataset with ground truth family labels," *Comput. Secur.*, vol. 124, Jan. 2023, Art. no. 102921.
[48] (2016). *A Tool for Reverse Engineering Android Apk Files*. [Online]. Available: https://github.com/hasherezade/pe-bear
[49] (2016). *A Tool for Reverse Engineering Android Apk Files*. [Online]. Available: https://practicalsecurityanalytics.com/pe-malware-machine-learning-dataset/
[50] H. Borchani, G. Varando, C. Bielza, and P. Larranaga, "A survey on multi-output regression," *Wiley Interdiscipl. Reviews: Data Mining Knowl. Discovery*, vol. 5, no. 5, pp. 216–233, 2015.

ALPER EGITMEN received the M.S. degree in computer engineering from Yıldız Technical University, Istanbul, Turkey. His current research interests include artificial intelligence, computer security, and digital signal processing

ALI GOKHAN YAVUZ received the Ph.D. degree in computer engineering from Yıldız Technical University, Istanbul, Turkey. He is currently a Professor and the Head of the Department of Computer Engineering, Turkish-German University. His current research interests include systems and network security, cloud computing, and big data.

SIRMA YAVUZ (Member, IEEE) received the Ph.D. degree in computer engineering from Yıldız Technical University, Istanbul, Turkey. She is currently a Professor and the Head of the Department of Computer Engineering, Yıldız Technical University. Her current research interests include artificial intelligence, neural networks, machines, and robotics.