Hindawi
Security and Communication Networks
Volume 2020, Article ID 6726147, 10 pages
https://doi.org/10.1155/2020/6726147

Research Article

Combat Mobile Evasive Malware via Skip-Gram-Based Malware Detection

Alper Egitmen,[1] Irfan Bulut,[2] R. Can Aygun,[3] A. Bilge Gunduz,[1] Omer Seyrekbasan,[1] and A. Gokhan Yavuz[1]

[1]Computer Engineering Department, Yildiz Technical University, Istanbul, Turkey
[2]OM Partners, Koralenhoeve 23, 2160, Wommelgem, Belgium
[3]University of California, Engineering VI, Los Angeles, CA 90095, USA

Correspondence should be addressed to Alper Egitmen; aegitmen@yildiz.edu.tr

Received 11 November 2019; Revised 19 February 2020; Accepted 28 March 2020; Published 20 April 2020

Academic Editor: Petros Nicopolitidis

Android malware detection is an important research topic in the security area. There are a variety of existing malware detection models based on static and dynamic malware analysis. However, most of these models are not very successful when it comes to evasive malware detection. In this study, we aimed to create a malware detection model based on a natural language model called skip-gram to detect evasive malware with the highest accuracy rate possible. In order to train and test our proposed model, we used an up-to-date malware dataset called Argus Android Malware Dataset (AMD) since the AMD contains various evasive malware families and detailed information about them. Meanwhile, for the benign samples, we used Comodo Android Benign Dataset. Our proposed model starts with extracting skip-gram-based features from instruction sequences of Android applications. Then it applies several machine learning algorithms to classify samples as benign or malware. We tested our proposed model with two different scenarios. In the first scenario, the random forest-based classifier performed with 95.64% detection accuracy on the entire dataset and 95% detection accuracy against evasive only samples. In the second scenario, we created a test dataset that contained zero-day malware samples only. For the training set, we did not use any sample that belongs to the malware families in the test set. The random forest-based model performed with 37.36% accuracy rate against zero-day malware. In addition, we compared our proposed model's malware detection performance against several commercial antimalware applications using VirusTotal API. Our model outperformed 7 out of 10 antimalware applications and tied with one of them on the same test scenario.

1. Introduction

Advancements in mobile device technology led developers to make rich content applications for different purposes such as *social media, health care, finance,* and *government*. Consequently, mobile device usage increased drastically, and malicious software (malware) developers turned their attention to mobile application markets [1–4]. Malware may have many different goals such as *encrypting personal data, using device resources (cryptocurrency), stealing sensitive information (financial information, pictures, contacts, etc.), converting victim's machine into a bot, and restricting access to critical services* [5].

In the application market arena, there are two main players, namely, Apple's App Store and Google's Play Store. According to the Statista report in [6], Android OS has the highest market share worldwide on mobile devices since 2011. Moreover, as of May 2017, Android has over two billion monthly active users, and as of December 2018 the Google Play store features over 2.6 million apps [6].

Also, it is a well-known fact that applications published in Google Play Store do not undergo a rigorous review process [7]. As a result, Android OS has become one of the OS most preferred by malware developers [5]. Therefore, the requirement of a robust malware detection approach for the Android Platform has become more imminent than ever before.

The widespread malware detection technique is often based on signature detection techniques. In signature detection-based techniques, the signature of an application, namely, its fingerprint, is compared against a database consisting of the fingerprints of known malware. So, these techniques are only limited to detect previously encountered malware and are vulnerable against zero-day malware and/or malware equipped with evasion capabilities. In order to deal with evasive and/or zero-day malware, it is essential to design an effective malware detection approach.

In recent years, security researchers have improved malware detection techniques by analyzing malware both statically and dynamically. In static analysis methods, important features such as application permissions, library dependence, and binary code structure are extracted from malware without requiring execution. On the other hand, in dynamic analysis, malware is run in a sandbox environment in order to observe and investigate its runtime behavior such as system calls, file access, network access, and registry activities. Dynamic analysis approach could potentially provide a wider view of malicious code capability referred to as modern malware. Modern malware either employs methods such as code obfuscation, call indirection, and string encryption to misguide static analysis tools or senses the runtime environment in order to apply countermeasures to avoid dynamic analysis.

In this study, we present an efficient malware detection method which counterbalances the evasion techniques employed by modern malware. Our proposed malware detection method is based on static analysis. We apply semantic analysis in contrast to syntactic analysis on the source code of a given application in order to classify it as benign or malicious. Our proposed model decompiles an application and extracts opcode sequences. Then, it uses a state-of-the-art NLP word embedding algorithm, which is called skip-gram, to generate the word embedding vectors for each unique opcode to obtain semantic relations among them. By using these word embedding vectors, we generate dense vectors to achieve a high-level representation of the opcode sequences. Then, we use these dense vectors as input to our detection model, which detects malware with an accuracy of 95.64%. Also, as a result of using dense vectors, we were able to decrease training time, memory usage, and processing power significantly.

The remainder of this manuscript is organized as follows: Section 2 gives information about related works. Section 3 describes our model and methods. In Section 4, the experimental results are discussed and analyzed. Finally, Section 5 concludes the work and describes future directions.

2. Related Work

Static malware analysis methods have been widely used for malware detection problem. Some of the security researchers who focused on static malware analysis used natural language processing approaches called n-gram or skip-gram for the feature extraction. These methods were applied to malware detection problem since malware analysis of bytecode/opcode sequences of malware is similar to text and document classification problem. Meanwhile, other researchers focused on more noticeable features such as API calls, call graphs, and user permissions for malware detection problem. In this study, our proposed model was built on a natural language processing approach called skip-gram. Because of that, we focused on existing NLP-based static malware analysis studies for the related work. It is important to note that even though we mention the detection accuracy rates for all of the previous studies given below, these results are not comparable with each other due to lack of a standard dataset to evaluate the proposed models' performances.

Moskovitch et al. extracted opcode n-gram from Windows executable binaries and applied feature selection methods like fisher score, gain ratio, and document frequency to determine the most critical n-gram terms [8]. They created a dataset based on 5677 malware and 20416 benign files. Their proposed method performs with 94.43% accuracy on a 2-gram feature-based dataset with Boosted Decision Tree (DT) method.

McLaughlin et al. proposed a method based on word embeddings obtained from opcodes to detect malware [9]. Convolutional Neural Network (CNN) classifier was used for malware detection. They have multiple datasets varying from 2,123 to 48,000 samples. The maximum accuracy of the malware detection rate is 98% with opcode 3-gram and small dataset. Nevertheless, the minimum accuracy of malware detection rate is 69% with large dataset, which contains 24,000 malware and 24,000 benign Android applications.

Karbab et al. proposed a method that extracts word embeddings from API call sequences, which were obtained from applications' assembly codes, to distinguish malware from benign [10]. The dataset consists of 33066 malware (20089 MalDozer, 5555 Drebin, and 1258 Malgenome) and 37627 benign applications. The maximum accuracy of the malware detection rate is 96%, and minimum malware detection rate is 95% on the entire dataset. The 99% accuracy was achieved by using only Drebin dataset with 5-fold cross-validation. This study focused on Android malware detection that relies on API call sequence-based features and CNN classifier.

Awad et al. focused on the effectiveness of word2vec feature extraction methods in malware detection area [11, 12]. Experiments done with 10,868 Windows malware contain nine different families. They extracted skip-gram and Continuous Bag-of-Words (CBOW) features from opcodes only and opcodes with abstract parameters. The maximum accuracy is 98% with CBOW with the window size of five, while the minimum accuracy is 76% with skip-gram with the same window size. This study was not focused on malware detection but effectiveness of the Natural Language Processing (NLP) methods on Windows malware classification.

Xu et al. proposed a method that includes two different detection stages and a user-provided threshold value. Applications are either classified at the first detection layer or passed to the second detection layer [13]. The first layer consists of XML feature vectors and uses Multilayer Perceptron (MLP) as a classifier. Meanwhile, the second layer consists of word embeddings' representation of application opcodes. Over 200 unique opcodes were grouped under 15 distinct categories for simplifying preprocessing cost. The dataset includes 62915 malware and 47525 benign applications. The 10-fold cross-validation was applied for the evaluation, and there was a validation set for the fine-tuning of deep learning hyperparameters. Their proposed model performed with an accuracy of 97.74%.

Yousefi-Azar et al. proposed an n-gram method to distinguish malware from benign. In their study, they used a malware dataset called PDF Share. The dataset consists of 10980 malicious PDF and 8999 benign PDF. The n-gram-based features were acquired from malware opcodes. Skip-gram and CBOW were created over n-gram instead of opcodes [14]. They applied a variety of machine learning algorithms to detect malware such as K-Nearest Neighbor (KNN), Extreme Learning Machine (ELM), Support Vector Machine (SVM) [15], and XGBoost (also known as GBDT, GBM) [16, 17]. The maximum accuracy of the malware detection rate is 98% yet, and the minimum malware detection rate is 92%.

In [18], Fan et al. especially focused on the sly malware. Both context and relation-based features were used to characterize malware. Different types of entities and their semantic relationships were modeled and a metagraph-based approach, which is called as heterogeneous information network (HIN), was proposed to describe these relationships. The authors also propose a new HIN embedding model meta-graph2vec to lower the training cost by forming low dimensional representations for the nodes in HIN. Consequently, a classification model was created by feeding SVM with low dimensional representations of Windows portable executable files. Moreover, an experimental study was carried out on real malware collections from Comodo Cloud Security Center to measure the performance of the proposed approach against already existing malware detection approaches. The accuracy of the model was obtained as 0.97. The authors name their malware detection software as Scorpion.

Wang et al. proposed a malware detection approach that uses URLs visited by applications to identify a malware [19]. First, skip-gram approach was used to represent the features as vectors; then a multiview neural network was used to create a malware detection model that emphasizes depth and width. Experimental studies were carried out to measure the performance of the model. In addition, a comparison between the detection results of the approach and wild applications was made with 10 popular antivirus scanners. When compared with other traffic-based studies, both F-Measure and accuracy are reported as 0.98. Moreover, this study was carried out using only the URLs in the http traffic, so the other URLs were considered out of scope.

Ye et al. proposed a real-time malware detection approach for Android malware by using heterogeneous graph [20]. In this study, first, runtime Application Programming Interface (API) call sequences were extracted from Android applications, and then their high-level semantic relationships in the ecosystem were analyzed. A heterogeneous graph (HG) structured for modeling was prepared to model different types of assets (i.e., application, API, device, signature, and connection) and the relationships between them. In order to efficiently classify the nodes (e.g., applications) in the created HG, training was carried out using the HG learning method. It was then given to a deep neural network classifier, taking the learned HG representations as input for real-time Android malware detection. The authors call their approach as AIDroid and report its accuracy as 0.99.

On the other hand, ANDRE [21], is an approach based on clustering to detect malware. It uses multiple sources of information, such as static code analysis results, meta-data of the applications, and raw tags from antivirus vendors of weakly tagged Android malware. Malware, whose malicious behavior is close to that of existing families on the network, was also classified using a three-tier Deep Neural Network (DNN). Unknown malware was clustered using a standard density-based clustering algorithm. The authors evaluated their approach using 5,416 ground-truth malware from Drebin and 9,000 malware from VirusShare consisting of 3324 weakly labeled malware. The maximum accuracy value was reported as 0.91 with the MLP classifier. In this study, obfuscated malware was not taken into consideration, and the white-list method was used to exclude common Android application third party libraries.

Ge et al. proposed an approach called AMDroid, which uses function call graphs (FCGs) that represent the behavior of applications and uses graph kernels to automatically learn the structural meaning of applications from FCGs [22]. The performance of AMDroid was evaluated on the Genome Project and experimental results show that AMDroid detected Android malware with 97.49% accuracy with the SVM classifier.

In [23], the authors used the API call chart as a graph representation of all possible execution paths that malware can follow during its execution. The embedded API call graphs converted into a low dimensional numerical vector feature set are then introduced to a deep neural network. Then, the similarity detection approach was effectively trained and tested for each binary function. Experimental results show that the accuracy of malware classification is at 98.86%. However, this study was carried out by using API call graphs, by focusing on the general success without taking into account the malware families according to the dynamic analysis principles.

Table 1 summarizes the related work studies based on the methods, dataset, and feature types used and gives obtained accuracies.

3. Evasion Techniques Used by Modern Malware

Modern malware uses a variety of techniques to fight against antimalware systems. Some of this malware apply techniques that help to avoid detection from antimalware systems, while the others develop and/or use some antianalysis methods for a specific malware detection system. Some techniques used by the malware are used to avoid detection while other techniques are used to prevent analysis. Most well-known and most frequently used evasion techniques used by modern malware are summarized below.

Table 1: Summary of related work.

Research	Acc. (%)	Dataset size	Methods	Feature types
[8]	94.4	26,093	Boosted DT	Opcode n-gram
[9]	69	48,000	CNN	Opcode word embedding
[9]	98	2,123	CNN	Opcode word embedding
[10]	96	70,693	CNN	Embedding
[11]	98	10,868	KNN	API call sequence opcode skip-gram CBOW
[13]	97.7	110,438	MLP + LSTM	APK XML opcode embedding
[14]	98	19,979	XGBoost	Skip-gram-based opcode n-gram
[18]	97	59,749	SVM	HIN metagraph2vec
[19]	98	—	Multiview NN	Skip-gram
[20]	99	190,696	HG learning	API call graph heterogeneous graph
[21]	91	14,416	Clustering + DNN	Skip-gram CBOW
[22]	97.49	2520	SVM	Skip-gram
[23]	98.86	58,139	DNN	API call graph skip-gram

(i) Renaming is one of the most basic techniques to get through antimalware software. In this technique the malware renames either itself or the names of the variables and functions used.

(ii) Repacking is an easy yet efficient evasion technique. In this technique, to look like inscrutable to others, malware authors either compress, encrypt, or rearrange some part of their malware [24]. When the malware is run, first it unpacks itself and then begins execution.

(iii) String encryption is a technique that malware uses to complicate the static analysis. Any significant string which could lead to the fingerprinting of the malware is encrypted using a custom encryption scheme. Static analysis will only be able to fingerprint the malware after proper decryption of such string.

(iv) Source code encryption is an obfuscation technique to hide malware from antimalware software. It can be applied either at program level or at class level. When applied at program level, the whole malware is encrypted; whereas when it is applied at class level, either selected classes or all classes are encrypted. Source code encryption will defeat the correct fingerprinting of the malware.

(v) Call indirection and reflection technique is a kind of transformation to manipulate the call graph of the application. In this way an automatic matching of the calls is avoided. A method call is converted to a call which is then invoked to the original call [25].

(vi) dynamic loading technique, malware externally loads data and/or code dynamically, from an external server at startup time. This external sourcing at runtime effectively evades a static analysis. Also, it is customary that the externally sourced pieces of data and code are also encrypted.

(vii) Resource obfuscation is another antidetection antianalysis technique malware that authors use. They obfuscate the way program resources, such as strings and graphical images, which are stored on disk, and then deobfuscate them at runtime so they can be used by the malware.

(viii) Antidisassembly is another technique used by malware authors. Anti disassembly aims at exploiting the inherent limitations of state-of-the-art disassembly techniques to hide code from malware analysis.

(ix) Antidynamic analysis method is specifically developed for avoiding dynamic analysis. There are two main categories regarding the determination of the run-time environment. The first category does not necessarily require to know if it is being run on analysis environment. These kinds of malware attacks wait for a fixed amount of time or event such as specific user interaction to activate malicious code. The second group has various methods to sense the environment that it is being run currently. This type of malware never activates malicious code if it senses that it is being run on analysis environment; thus it is identified as a benign application. It can obtain this information by various methods such as accessing global variables (contacts, picture counts, etc.), measuring inconsistency of cache memory access times, or even finding a side channel which leaks sensitive information about analysis environment that should not be known by malware.

4. Raw Datasets

When the datasets in the studies mentioned in the related work section were examined, two basic problems appeared. First, most publicly available datasets are outdated. Second, these datasets contain limited amount of different malware families. Modern malware has complex evasion techniques that give them robustness against static and dynamic analysis. Because of that, it is important to use datasets that contain complex malware families to be able to evaluate the performance of malware detection model accurately. Another problem is that to the best of our knowledge there is no public benign dataset available. In this study to tackle with these problems we created our own malware detection dataset comprised AMD Argus Lab Malware Dataset, Comodo Benign Dataset [26], and hand-crafted benign applications from Google Play Store. The total size of dataset is about 107 GB. Argus Lab Malware Dataset was publicly released in 2017 by Argus Laboratories [27]. It contains 71 different malware families and a large set of unique malware instances for each family. Total of 24,553 Android Packages (APKs) was collected between 2010 and 2017. Behavioral analysis was applied to each family and several attributes were obtained such as installation method, composition, activation, antianalysis methods, monetization, information stealing, and detailed graphs that show how actually the malware works. To extract antianalysis attribute, each family was tested against renaming, string encryption, dynamic loading, native payload, and several dynamic analysis evasion methods to explore their capabilities. Having this detailed meta-data information, we can tell robustness of our model against such evasion techniques. The dataset is very large and detailed, which fills modern and large dataset gap for researchers. Size of the APKs in this dataset varies from 10 KB to 48 MB. In this dataset, we were able to obtain opcodes from 24.304 samples using Apktool. The remaining samples were discarded from the final dataset due to difficulties in decompilation phase [27]. On the other hand, ISL Benign Dataset contains 16,630 benign applications. Half of them were collected from Google Play Store in different categories, and the other half were obtained from Comodo Security Group. The half obtained from Google Play Store contains applications from 34 different categories, as classified by Google, whereas the dataset provided by Comodo Security Group contained applications of only 12 different categories. The applications selected from Google Play Store were all marked as benign by Google Play Protect. To be on the safe side, each sample in the benign dataset was validated for its benignity by using VirusTotal API. Via VirusTotal API each sample was scanned with the top 15 antimalware software and only samples which were indicated as benign by all the antimalware software were added to the benign dataset. ISL Benign Dataset contains applications from completely different categories and varies in size from 2 KB to 50 MB.

5. Preprocessing

Our dataset consists of raw APK archives. APK is a specific archive format for Android applications. It contains Dalvik Executable Files (DEX), resources, user permissions, metadata, and various configuration files. DEX files are like class files in Java but they are converted to DEX format since a different bytecode format is used in Dalvik or Android Runtime (ART) Virtual Machine. Opcode sequences were obtained from DEX files belonging to each APK within the dataset and then they were written to respective text files. This process caused a significant growth in the size of the dataset, so, we encoded the opcodes using Huffman encoding [28]. Consequently, the amount of space required to store the preprocessed APK files was reduced and as a result much less IO operations execution times for training, validation, and testing were significantly diminished. Extracting of opcode-based features from APK files consists of several stages. As depicted in Figure 1 Apktool was used to obtain corresponding DEX files for each APK file [29]. Then, each DEX file was converted into Android opcode sequences(smali) using baksmali decompiler [30]. The resulting decompiled files were then processed with Huffman encoder in order to represent the opcodes space efficiently. Finally, the files were merged into a single file. This process was repeated for each APK instance within the dataset.

6. Malware Detection Model

We consider opcode sequences from Android applications as artificial language. In this language, unique opcodes are language words, functions as sentences and whole sequence as part of the corpus. To build semantic relations between opcodes, we used the skip-gram method which is a part of the Word2Vec model [31].

Word2vec is a group of related models that are used to produce word embeddings. These models are shallow, two-tier neural networks that are trained to reconstruct linguistic contexts of words. Word2vec takes a large text corpus as input and produces a vector space, typically of several hundred dimensions, for each unique word in the corpus.

6.1. Skip-Gram. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located in close proximity to one another in the space. Skip-gram is a feature extraction model that is used to predict the potential words that might come before and after a selected word. We refer to this selected word as a target and words around it in certain window size as a context. The skip-gram model is trained in an unsupervised fashion for each target and context tuples. After the training part is completed, the corresponding weights are considered as a word embedding vector for each word in the vocabulary. In Figure 2 this process is explained.

6.2. Apk2Vec. Applications in our dataset have a great diversity in terms of opcode sequence length. This creates an inconsistent input layer dimension problem for the training of machine learning-based models. One possible solution to this problem is to put padding to input layers for all samples in the dataset. This solution may work in spoken languages where the sentence or document lengths have significantly low variance. However, in our dataset, input layer dimensions vary from several hundreds to three millions. So, padding creates a sparse dataset problem. To overcome this problem, we used a simple but efficient method, which is called "Apk2vec," to describe every smali file with a fixed size vector. We represent every opcode in the smali file with its corresponding embedding vector; then we sum embedding vectors for each opcode. Then we calculate the average embedding vector. We consider this vector as fingerprint of the APK (Figure 3). We repeated this process for each APK archive in our dataset to create our final malware detection

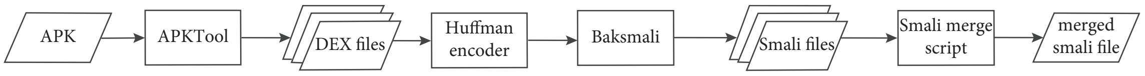

Figure 1: Preprocessing of an Android application.

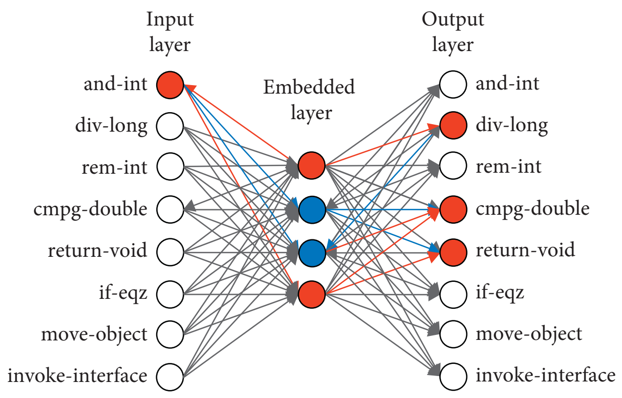

Figure 2: Skip-gram learning process for target word and-int and context words div-long, cmpg-double, and return-void. Input is one hot vector of target word while output tries to predict correct context words. Dense vector for each word is weighted from corresponding input neuron to embedded layer neurons.

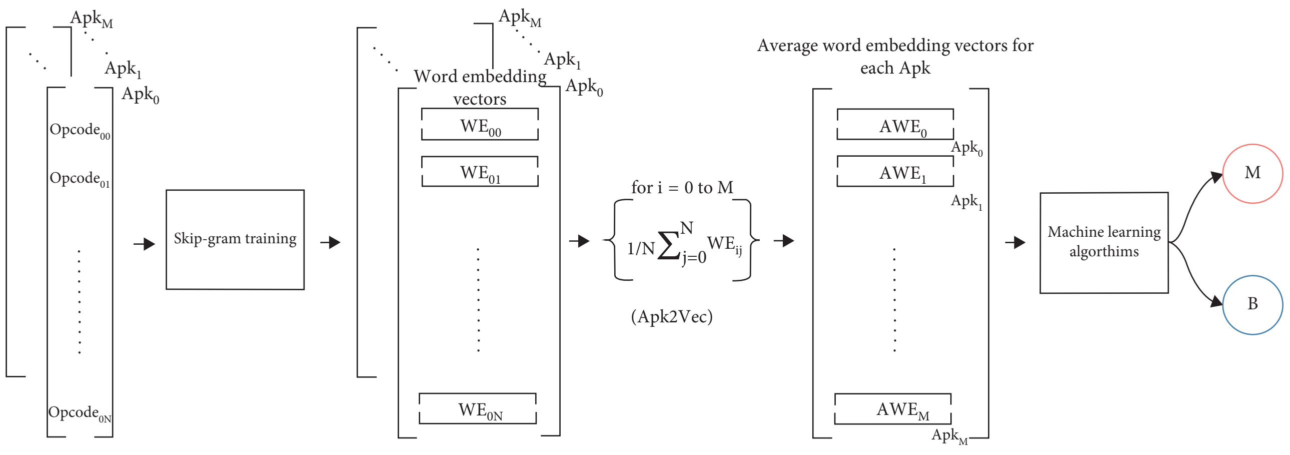

Figure 3: Proposed malware detection model. The output M is used as malware; B is used as benign.

dataset, which is suitable for machine learning-based classification.

7. Experiments and Results

Application of skip-gram method on opcodes is a new concept. Skip-gram was designed to create models for speaking languages, which has very large size of vocabulary but small sentence size. But opcode sequences have completed opposite characteristics. We have found 219 unique actively used opcodes in the complete Android opcode set, which contains 238 different opcodes. We consider opcodes as words and functions as sentences of our language model. This made us have significantly long sentences with a very small vocabulary. Thus, using prevalent hyperparameters for skip-gram training in natural language classification may hurt our final detection performance. To eliminate this, we conducted a prior classification test to obtain the best performed hyperparameters. For this test, we created a small dataset containing only 1500 malware and 1500 benign applications from our large dataset pool. During the compilation of the instances for the 1500 malware, we paid special attention to include instances from all of the 71 malware families with the homogeneous distribution. As to the 1500 benign samples they were randomly selected from the benign dataset. There exist no families among benign instances.

We then split this small dataset into the test and training dataset with ratio of 70% and 30%, respectively. We tried to create a balanced dataset that contains at least one instance from every malware family with window sizes of 2 and 3 and word embedding sizes of 50, 300, and 500. *Random forest* [32], *decision tree* [33], *Naive Bayes*, and *SVM* were used as classifiers.

7.1. *Malware Detection and Zero-Day Performance.*

We explored two different scenarios to test our model's performance and its robustness against zero-day malware. For

Table 2: Subset of dataset trained for tuning of different hyperparameters of skip-gram, window size, and embedding size.

		J48		Naïve Bayes		Random forest		SVM	
Window size	—	2	3	2	3	2	3	2	3
Embedding size	50	79.77%	78.91%	60.51%	60.44%	83.89%	83.56%	60.01%	59.78%
	300	81.93%	81.50%	60.81%	62.37%	87.08%	87.14%	59.61%	60.04%
	500	82.63%	82.82%	61.60%	65.99%	86.51%	88.54%	59.21%	57.25%

Table 3: Dataset samples split 30% and 70% for test and training, respectively, for scenario 1 and scenario 2. 1999 samples from 21 families are excluded from training set for simulating zero-day attack.

		Scenario 1	Scenario 2
Training set	Malware	16982	15593
	Benign	11641	11641
Test set	Malware	7322	6712
	Benign	4988	4988
Zero-day	Malware	—	1999

Table 4: Scenario 1 test results for different machine learning methods. Ensemble 1 is majority voting of all other methods shown in this table. Random forest bested other methods with 95.64% accuracy.

	Malware			Benign			Total accuracy
	Precision	Recall	F1 measure	Precision	Recall	F1 measure	
SVM	86.5	79.7	82.9	73.2	81.7	77.2	80.48%
Random forest	97	95.6	96.3	93.7	95.7	94.7	95.64%
Decision tree	91.9	93.6	92.7	90.3	87.9	89.1	91.25%
Random subspace	95.7	94.4	95.1	92	94.8	92.9	94.18%
SGD	82.9	93.4	87.8	88.8	71.7	79	84.58%
KNN	93.4	96.9	95.1	95.2	90	92.5	94.09%
Ensemble 1*	95	96.5	95.7	94.7	92.5	93.7	94.88%

the first scenario, we tried to create the training and test datasets to contain at least one instance from each of the 71 malware families, whereas for the zero-day performance 21 malware families were left out during the composition of the corresponding datasets. Since we have a large and well-balanced number of instances, we opted for 70% to 30% balance among training and test datasets instead of using cross-validation. We created the corresponding training datasets to contain at least one instance from each included malware family. Later we preprocessed the training and test datasets to obtain Huffman encoded opcode sequences for the instances. This encoding resulted in a reduction of more than 80% of the raw datasets which approximately corresponds to 500 GB. For each scenario, we considered the training dataset as the corpus to be used in the skip-gram model and generated an embedding matrix with a window size of 3 and an embedding size of 500. For the window size, we tried the values two and three; for the embedding size we tried the values 50, 300, and 500. The obtained accuracies for the selected classifiers with varying window and embedding sizes are summarized in Table 2. Consequently, the aforementioned values for the respective sizes were chosen as they resulted in the best accuracy values. Using the embedding matrix obtained from the previous step, we ran Apk2vec algorithm for each test scenario and obtained the respective fixed size vector representations. These vectors were used as inputs to the machine learning classifiers. We used decision tree (DT), random forest (RF), 1-NN neighborhood, random subspace, SVM, and gradient descent. The instances from the left-out 21 malware families were used to test the zero-day performance using the model generated with the 50 malware families as a part of scenario 2. The number of instances and their distribution for each scenario are given in Table 3.

7.2. Results. For performance evaluation, F1 measure, precision, recall, and accuracy metrics were calculated. Since, our overall dataset contained balanced number of benign and malware instances, we chose the accuracy metric as the decisive criteria to reflect the performance of our proposed model. Tables 4 and 5 summarize the obtained accuracy values for the generated models. For both of the models, RF algorithm gave the best precision with 97% and 97.4%, respectively. The 1999 instances from the left-out malware families were tested for malwareness using the RF-based model from the second scenario. Our model successfully detected 747 instances as malicious.

We also compared our model's performance to the performances of top 11 reliable antimalware software applications. As these antimalware software use accuracy as the performance metric, we compared our accuracy value to their corresponding values. The results of the comparison are given in Table 6, which shows that our model outperformed seven out of 11 commercial software applications and tied in with one.

Table 5: Dataset used in scenario 2 contains 21 less malware families which was excluded for zero-day testing; this caused small increase in detection performance. RF bested other methods with 96.12% accuracy.

	Malware			Benign			Total accuracy
	Precision	Recall	F1 measure	Precision (%)	Recall (%)	F1 measure (%)	
SVM	87.6	79.8	83.5	75.8	84.8	80	81.94%
Random forest	97.4	95.8	96.6	94.4	96.6	95.5	96.12%
Decision tree	92.5	93.9	93.2	91.6	89.7	90.6	92.04%
Random subspace	96.4	94.6	95.5	92.9	95.2	94	94.86%
SGD	70.4	97.4	81.7	92.7	45	60.6	75.04%
KNN	93.5	97.1	95.3	95.9	90.9	93.4	94.48%
Ensemble 1*	94.8	96.9	95.8	95.7	92.8	94.2	95.15%

Table 6: Comparison of our model against commercial antimalware software; we used VirusTotal API to evaluate our test dataset. Our proposed method bested 7 of 10 commercial antimalware software applications.

Commercial software	True positive	False negative	Accuracy
Avast	2900	4422	0.39
AVG	2902	4420	0.39
Avira	6797	525	0.92
Comodo	6367	955	0.86
Eset-Nod32	7304	18	0.99
F-secure	6311	1011	0.86
Kaspersky	4144	3178	0.56
McAfee	7265	57	0.99
Microsoft	4075	3247	0.55
Sophos	7074	248	0.96
Symantec	7225	97	0.98
Proposed model	7066	256	0.96

Table 7: Robustness of our model against evasion methods; this table shows malware families, their evasion capabilities, and their sample counts in test set. Final column shows our model detection performance of corresponding malware family.

	Evasion methods					Results	
Family name	Renaming	String encryption	Dynamic loading	Native payload	Antidynamic analysis	Total count	Accuracy (%)
Airpush	✓					2353	95
Andup	✓	✓				14	92
BankBot	✓	✓	✓		✓	195	99
Bankun					✓	21	90
Boqx				✓		65	75
Boxer	✓	✓				14	100
Cova	✓					6	100
Dowgin	✓	✓	✓		✓	1015	91
DroidKungFu	✓	✓		✓		164	98
FakeAngry	✓	✓				3	66
FakeDoc	✓					7	100
FakeInst	✓	✓				651	100
FakePlayer	✓					7	85
FakeUpdates	✓	✓				2	0
Finspy	✓				✓	3	100
Fobus		✓	✓		✓	2	100
Fusob	✓	✓	✓			383	100
GingerMaster	✓	✓				39	97
GoPro	✓	✓			✓	11	63
Gumen				✓	✓	44	88
Koler	✓				✓	21	100
Ksapp					✓	11	90
Kuguo	✓					360	98
Kyview	✓	✓				53	86
Leech	✓	✓	✓		✓	39	100
Lotoor	✓			✓		95	96

Table 7: Continued.

	Evasion methods					Results	
Family name	Renaming	String encryption	Dynamic loading	Native payload	Antidynamic analysis	Total count	Accuracy (%)
Minimob	✓					61	88
Mseg	✓					71	53
Mtk	✓	✓	✓			21	100
Obad	✓	✓			✓	3	100
Opfake		✓				3	66
Ogel	✓			✓		2	100
Roop	✓					155	99
RuMMS	✓	✓	✓			121	100
SlemBunk		✓	✓	✓		52	100
Simplelocker	✓					48	100
SmsKey	✓					50	98
Stealer	✓					8	100
Svpeng					✓	4	75
Tesbo	✓	✓				2	100
Triada	✓	✓	✓		✓	63	95
UpdtKiller	✓			✓		8	100
Utchi	✓					4	100
Viking Horde				✓		3	33
Winge	✓					6	16
Youmi	✓					390	97
Zitmo	✓					8	87
Ztorg	✓	✓	✓			6	100
Total count	6462	2856	1796	433	1432	6667	—
Accuracy (%)	95	96	94	93	92	—	95 (total)

Moreover, we also evaluated the performance of our model in terms of its robustness against evasive malware. In the first scenario 48 out of 71 malware families were marked as evasive by the dataset owners, which resulted in 6667 instances of evasive malware in the test dataset. The accuracy distribution of our model based on the evasiveness of the malware is given in Table 7. It can be seen that our proposed model was able to determine 95% of all evasive instances within the test dataset.

8. Conclusions

In this research, we proposed a novel and an efficient way to classify modern Android malware. In our proposed method we approached Android software as an artificially generated text but applied skip-gram technique, which was composed for NLP, to extract useful features.

We also proved that NLP-based static analysis approach to the application source codes has promising results. We used newly published Argus AMD dataset. This dataset provides malware behavior by their families such as evasion method. This detailed information let us evaluate the robustness of our model against certain evasion methods using machine learning algorithms. In conclusion, an accuracy of 95.64% was achieved without having to run target application and risking system stability.

Data Availability

The dataset is provided by Comodo and AMD Argus Lab. AMD Argus Lab dataset is public; however, Comodo dataset is private, and authors do not have the sharing privilege.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The authors would like to thank both Comodo Corporation and AMD Argus Labs for providing them with necessary datasets. Thanks are due to VirusTotal, which allows them to use their services to verify their dataset. Special thanks are due to BURAK TAHTACI and ONUR OZTUNC for their efforts in providing the working environment.

References

[1] V. Chebyshev, "Mobile malware evolution 2018," 2019, https://securelist.com/mobile-malware-evolution-2018/89689/.

[2] gdatasoftware, "Mobile malware evolution 2017," 2017, https://www.gdatasoftware.com/blog/2018/03/30610-malware-number-2017.

[3] gdatasoftware, "Mobile Trends 2017," 2017, https://www.gdatasoftware.com/blog/2018/03/30610-malware-number-2017.

[4] Statista.com, "Mobile android version share worldwide 2018-2019 | Statistic," 2019, https://www.statista.com/statistics/921152/mobile-Android-version-share-worldwide/.

[5] R. Benzmüller, "Malware statistics in 2017," 2017, https://www.gdatasoftware.com/blog/2018/03/30610-malware-number-2017.

[6] Statista, "Number of available applications in the google play store," https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.

[7] Google, "Google play store," https://play.google.com/store.

[8] R. Moskovitch, C. Feher, N. Tzachar et al., "Unknown malcode detection using opcode representation," *European Conference on Intelligence and Security Informatics*, pp. 204–215, Springer, Berlin, Germany, 2008.

[9] N. McLaughlin, J. Martinez del Rincon, B. Kang et al., "Deep android malware detection," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ACM, New York, NY, USA, pp. 301–308, March 2017.

[10] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "MalDozer: automatic framework for android malware detection using deep learning," *Digital Investigation*, vol. 24, pp. S48–S59, 2018.

[11] Y. Awad, M. Nassar, and H. Safa, "Modeling malware as a language," in *Proceedings of the 2018 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, Kansas City, MO, USA, May 2018.

[12] Dav, "word2vec.," 2019, https://github.com/dav/word2vec.

[13] K. Xu, Y. Li, R. H. Deng, and K. Chen, "Deep refiner: multi-layer android malware detection system applying deep neural networks," in *Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 473–487, IEEE, London, UK, April 2018.

[14] M. Yousefi-Azar, L. Hamey, V. Varadharajan, and S. Chen, "Learning latent byte-level feature representation for malware detection," in *International Conference on Neural Information Processing*, pp. 568–578, Springer, Berlin, Germany, 2018.

[15] S. Amarappa and S. Sathyanarayana, "Data classification using support vector machine (svm), a simplified approach," *International Journal of Electronics and Computer Science Engineering*, vol. 3, pp. 435–445, 2014.

[16] D. Opitz and R. Maclin, "Popular ensemble methods: an empirical study," *Journal of Artificial Intelligence Research*, vol. 11, pp. 169–198, 1999.

[17] H. Robbins and S. Monro, "A stochastic approximation method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951.

[18] Y. Fan, S. Hou, Y. Zhang, Y. Ye, and M. Abdulhayoglu, "Gotcha-sly malware! scorpion a metagraph2vec based malware detection system," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Association for Computing Machinery, London, UK, pp. 253–262, July 2018.

[19] S. Wang, Z. Chen, Q. Yan et al., "Deep and broad url feature mining for android malware detection," *Information Sciences*, vol. 513, pp. 600–613, 2020.

[20] Y. Ye, S. Hou, L. Chen et al., "Out-of-sample node representation learning for heterogeneous graph in real-time android malware detection," in *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pp. 4150–4156, AAAI Press, Macao, China, August 2019.

[21] Y. Zhang, Y. Sui, S. Pan et al., "Familial clustering for weakly-labeled android malware using hybrid representation learning," *IEEE Transactions on Information Forensics and Security*, p. 1, 2019.

[22] X. Ge, Y. Pan, Y. Fan, and C. Fang, "Amdroid: Android-malware detection using function call graphs," in *Proceedings of the 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pp. 71–77, IEEE, Sofia, Bulgaria, July 2019.

[23] A. Pektaş and T. Acarman, "Deep learning for effective android malware detection using api call graph embeddings," *Soft Computing*, vol. 24, pp. 1027–1043, 2020.

[24] J. Saxe and H. Sanders, *Malware Data Science:Attack Detection and Attribution*, No Starch Press, San Francisco, CA, USA, 2018.

[25] M. Ikram, P. Beaume, and M. A. Kaafar, "DaDiDroid:an obfuscation resilient tool for detecting android malware via weighted directed call graph modelling," 2019, https://arxiv.org/abs/1905.09136.

[26] Comodo, "Comodo anti-malware database," 2019, https://www.comodo.com/home/internet-security/updates/vdp/database.php.

[27] ArgusLab, "Android malware dataset," 2016, https://amd.arguslab.org/.

[28] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[29] ApkTool, "Apktool - a tool for reverse engineering 3rd party, closed, binary android apps," 2019, https://ibotpeaches.github.io/Apktool/.

[30] J. Freke, "Baksmali is an assembler/disassembler for the dex format," 2019, https://github.com/JesusFreke/smali.

[31] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, https://arxiv.org/abs/1301.3781.

[32] L. Breiman, "Random forests," *Machinelearning*, vol. 45, pp. 5–32, 2001.

[33] J. R. Quinlan, *C4.5: Programs for Machine Learning*, Elsevier, Amsterdam, Netherlands, 2014.