Hindawi
Security and Communication Networks
Volume 2022, Article ID 3710968, 15 pages
https://doi.org/10.1155/2022/3710968

WILEY | Hindawi

*Research Article*

# Op2Vec: An Opcode Embedding Technique and Dataset Design for End-to-End Detection of Android Malware

**Kaleem Nawaz Khan**,[1] **Najeeb Ullah**,[1] **Sikandar Ali**,[2] **Muhammad Salman Khan**,[3] **Mohammad Nauman**,[4] **and Anwar Ghani**[5]

[1]*Department of Computer Science, University of Engineering and Technology Mardan, Mardan, Pakistan*
[2]*Department of Information Technology, The University of Haripur, Haripur 22620, Khyber Pakhtunkhwa, Pakistan*
[3]*Department of Electrical Engineering, College of Engineering, Qatar University, Doha, Qatar*
[4]*Department of Computer Science, National University of Computer and Emerging Sciences, Peshawar, Pakistan*
[5]*Department of Computer Science and Software Engineering, International Islamic University Islamabad, Islamabad, Pakistan*

Correspondence should be addressed to Kaleem Nawaz Khan; kaleemnawaz@uetmardan.edu.pk, Najeeb Ullah; najeeb@uetmardan.edu.pk, Sikandar Ali; sikandar@uoh.edu.pk, and Anwar Ghani; anwar.ghani@iiu.edu.pk

Received 25 February 2022; Revised 18 April 2022; Accepted 25 April 2022; Published 19 May 2022

Academic Editor: Shah Nazir



Android is one of the leading operating systems for smartphones in terms of market share and usage. Unfortunately, it is also an appealing target for attackers to compromise its security through malicious applications. To tackle this issue, domain experts and researchers are trying different techniques to stop such attacks. All the attempts of securing the Android platform are somewhat successful. However, existing detection techniques have severe shortcomings, including the cumbersome process of feature engineering. Designing representative features require expert domain knowledge. There is a need for minimizing human experts' intervention by circumventing handcrafted feature engineering. Deep learning could be exploited by extracting deep features automatically. Previous work has shown that operational codes (opcodes) of executables provide key information to be used with deep learning models for the detection process of malicious applications. The only challenge is to feed opcodes information to deep learning models. Existing techniques use one-hot encoding to tackle the challenge. However, the one-hot encoding scheme has severe limitations. In this paper, we introduce (1) a novel technique for opcodes embedding, which we name Op2Vec, and (2) based on the learned Op2Vec, we have developed a dataset for end-to-end detection of Android malware. Introducing the end-to-end Android malware detection technique avoids expert-intensive handcrafted feature extraction and ensures automation. Some of the recent deep learning-based techniques showed significantly improved results when tested with the proposed approach and achieved an average detection accuracy of 97.47%, precision of 0.976, and $F1$ score of 0.979.

## 1. Introduction

Mobile technology has shown exponential growth in the recent past. Mobile devices are the best source to manage our day-to-day communications. These mobiles accompany us in all our movements. The use of these devices allows us to handle most of our very important activities, social networking, payments, and banking, with ease. Due to the high growth rate, mobile platforms are highly targeted and severely infected with malicious applications [1]. Attackers look for the possibility of exploiting mobile platforms through various techniques. There are cybercriminals as well as hackers, sponsored by states, doing research in order to find schemes for possible attacks against mobile platforms for their better interest. According to International Data Corporation (IDC) [2], mobiles have surpassed PCs in terms of preferred devices that can be used to access the Internet and other possible services. IDC also states that the number

of mobile users will cross the 91 million mark over the coming four years.

Malware is a short form used for malicious software. It is a program or software on any system that is not intentionally installed by the end-user or system administrator. There are various types of malware for different tasks and purposes [3]. The behavior of malware can range from being a very simple piece of annoyance, such as pop-up advertisements to severe actions which may be much more damaging and harmful [4], such as stealing important systems' passwords or secret data and other more severe actions. They may be used for infecting other machines having very confidential and secret information, over the network [5].

Among mobile platforms, Android is one of the most prevalent platforms for smartphones nowadays. It has seen exponential growth with a market share of 82.6% and has several millions of mobile applications in various markets [2]. It is a very rich platform in terms of the availability of various functionalities to its users. Unfortunately, it has been observed that smartphones with the Android operating system are targeted more often than any other platform by security attackers [4], and it is very severely infected by malicious software. Unlike other mobile platforms, Android allows easily installing applications from sources without clear verification, such as third-party markets, whose sole purpose is to bundle and distribute mobile applications with malwares, assisting attackers in different kind of tasks [6]. According to a report [7], the number of Android malicious applications will cross 3.8 million mark at the end of this year. Keeping this evidence in mind, there is a need for techniques and solutions to limit the production of malwares on different Android markets. Large body of research is involved to overcome the situation [8, 9]. Researchers are trying to find out smart ways for automated detection of malicious applications.

Android applications can be analyzed in two ways: either performing static analysis [10] or dynamic analysis [11]. In static analysis, the application is studied in its static position. Its behaviors, i.e., code patterns, requested permissions, relationships with other applications, intent filters, and other features, are analyzed. On the other hand, in dynamic analysis, the application is studied and analyzed during its running state. Dynamic aspects, such as observation of system calls, dynamic loading of the code segment, and invocations of API calls, are analyzed. Dynamic analysis is performed mostly in a controlled environment that is named a sandbox [12]. All the relevant operations of the state of the execution are monitored, such as sending SMS messages, storage reading, and connection to remote servers.

The conventional Android malware detection pipeline is to take Android applications and uses domain expertise to extract handcrafted features from a set of applications. Dynamic and static features are extracted for dynamic and static analyses, respectively. The features are then used to train machine learning algorithms to produce trained models to classify and detect Android malware. Common classifiers used for Android malware detection are support vector machine (SVM), decision tree (DT), *k*-nearest neighbors (KNN), random forest (RF), neural networks (NN), and *k*-means clustering. Recently, deep neural networks are getting attention for Android malware detection. Studies such as Droid-Sec [13], DeepDetector [14, 15], and [16] are using a deep learning approach to detect Android malware. Unlike handcrafted feature extraction for conventional machine learning algorithms, deep learning has a very strong and unique approach to automatically extract deep features and learn classification patterns.

All the conventional machine learning and the deep learning techniques studied in the existing literature work well with reasonable accuracy, but the problem is that these techniques rely on engineered handcrafted features. Even the deep learning techniques are trained with handcrafted features. Features engineering is a cumbersome and a very lengthy process, which requires domain knowledge. The feature engineering process is depicted in Figure 1. Domain knowledge and domain experts are required to perform brainstorming of features to decide what features to create. The created features are then tested with the experimentation model. Features are tuned where required and the complete feature engineering cycle is repeated if necessary. In most cases, malware designers are required to design the representative features. The domain experts and the available known malware designers are limited in number. That is why there is a need for making a system that can replace this lengthy and cumbersome features engineering process and incorporate end-to-end learning. The major problem is that we do not have any dataset publicly available for deep learning algorithms to learn end-to-end, i.e., extract deep features instead of designing handcrafted features. In end-to-end learning, the algorithm learns deep features instead of taking engineered features. So, the gap in the current research is to develop a dataset for end-to-end learning of Android malware and allow deep learning algorithms to be trained on the dataset and detect Android malware with minimum human expert intervention. Another problem in the existing solutions is that they use one-hot encoding to feed information to deep learning models. One-hot encoding creates severe problems and sometimes it becomes infeasible to be used with deep learning techniques. These limitations are discussed in detail in the coming sections. We need to devise an alternative that outperforms one-hot encoding.

Our first contribution concerns the development of a novel dataset for end-to-end detection of Android malware. The dataset can be used to learn useful patterns and information from the Android source code. We have not only developed the dataset but have also presented the design process and techniques involved in the dataset development. End-to-end learning minimizes human experts' intervention in designing/developing representative features and circumventing the handcrafted feature extraction process. The second contribution of this study is learning Op2Vec. Op2Vec learning process employs a machine-learning algorithm to learn meaningful vector representations from opcodes of Android source files. All the limitations of existing embedding techniques motivated us to develop a novel encoding technique which we term "Op2Vec." After learning Op2Vec and dataset development, we also validate

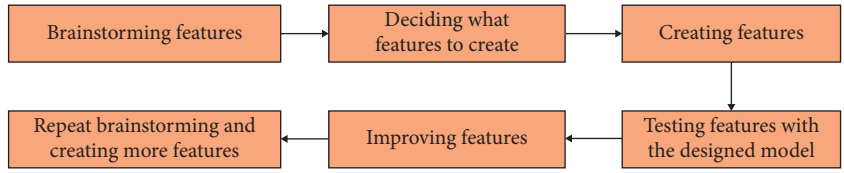


Figure 1: Features engineering process.

the dataset by feeding it to a deep neural network. Based on the above discussions, the main contributions are summarized as follows:

(1) Op2Vec for learning opcode-vector embedding
(2) Development of a dataset for end-to-end learning based on learned Op2Vec embedding and feeding the developed dataset to deep learning models for learning patterns and insights (circumventing feature engineering)

In this section, after a detailed introduction to the problem area and different techniques, the relevant background of Android malware detection and motivation for the proposed approach is discussed in Section 2. Section 2 focuses on malware detection techniques along with details of machine learning models, more importantly, a brief introduction, and the working paradigm of deep learning. Moreover, in the last part of the section, the word embeddings technique, its specifications, and its relevance to our novel Op2Vec technique are discussed. In Section 3, the methods of our research and the subtasks of the proposed approach are listed along with a brief discussion; more importantly, the word embedding model, skip-gram, is presented in depth with its working fashion. Section 4 discusses the experiments performed and the results of the experiments along with a detailed analysis. This section also covers how to use the designed dataset with deep neural networks for end-to-end learning. Section 5 provides a comparison of our proposed dataset with other state-of-the-art datasets and discusses how the use of Op2Vec makes it better than existing datasets. Section 6 concludes the paper and provides further extensions in this part of the research.

## 2. Background and Motivation

*2.1. Android Malware Analysis.* Due to the severity of Android malware, there are hundreds of approaches for its detection and classification. One approach is to perform a static analysis of the Android application. In static analysis, the executable code of the Android applications is examined to determine the data flow, control, and representative pattern without running the executables. Another approach is to examine an Android application through dynamic analysis. In dynamic analysis, the execution of an application is monitored by inspecting execution behaviors. Both static and dynamic analysis studies are discussed in the next two subsections.

*2.2. Dynamic Analysis Techniques for Android Malware.* Due to the complexity of dynamic analysis, there are very few techniques for conducting dynamic analysis for Android malware detection. A very popular technique is DroidScope [17]. It is an Android malicious applications' analysis engine, based on emulation, which performs dynamic analysis of Android applications. Its specialty is to reconstruct both OS and Java level semantics seamlessly and simultaneously. Riskranker [18] is another approach that ensures accurate and very scalable detection of zero-day Android malware. It dynamically analyzes whether a particular application exhibits dangerous behavior. Another similar technique DroidRanger [19] performs a behavioral footprinting scheme, based on requested permissions, for the detection of new samples of unseen Android malware families, and later applies a heuristics-based filtering approach for the identification of certain inherent behaviors of unseen malware application families.

Android malware detection is based on system calls [20], where automatic classification is performed based on tracking system calls. DroidScribe [21] is a very recent approach that focuses on dynamic analysis, i.e., runtime behavior. This approach shows how machine learning algorithms can be used to automatically classify Android malware into different malware families by just observing their runtime behavior. It observes that, on Android systems, system calls solely do not provide sufficient semantic content to make classification; that is why it also uses a lightweight VM introspection to reconstruct interprocess communication on the Android system for effective analysis. All the discussed techniques in this section have used engineered handcrafted features which are specifically designed for dynamic analysis.

*2.3. Static Analysis Techniques for Android Malware.* Static analysis is relatively simple and scalable. There are several techniques that use static analysis for Android malware detection. A set of static features is designed using domain expertise. The features are then utilized for the detection of Android malware. An approach, DroidAPIMiner [22], statically mines API-level features for the robust detection of malware on Android systems. It aims to provide a lightweight and robust classifier in order to evade Android malware installation. Another similar technique Droidminer [23] uses a behavioral graph to embed abstract malware program logic into a sequence of threat patterns and then applies machine learning techniques to identify and label elements of the graph that match already extracted threat modalities. A semantic-based technique [24] is responsible for the classification of Android malware through dependency graphs. It extracts a dependency graph, i.e., a weighted contextual API graph, as program semantics for the construction of feature sets. Another study reveals that

malicious application treats sensitive data differently from benign applications. This feature is used to design a useful technique [25] for the identification of malware applications. Benign Android applications are mined for their data flow from sensitive sources, and then, these flows are compared against those, found in malicious applications, to detect similarities.

DREBIN [26] is a lightweight method for Android malware detection. It enables the identification of malware applications on smartphones directly. It extracts as many features of an Android application as possible, and later, the extracted features are embedded into a joint vector space in such a fashion that typical patterns, which are indicative of malicious applications, can be automatically detected and can be used to explain decision making. DREBIN has shown an effective rate of detection of malicious applications on its own dataset. They have designed a dataset and a support vector machine (SVM) classifier is used for training on the same dataset.

A model presented in [27] is an *n*-grams-based technique. Sequential features, based on n-grams, are extracted from files' content. It determines patterns of sequential *n*-grams and then calculates statistics for the extracted pattern, and in the last, the classifier is trained to classify malware in different malware families. They have studied three classifiers for this task, i.e., SVM, C4.5, and multilayer perceptron. A method proposed in [28] uses three metrics, the weighted sum of permissions' subset, a set that consists of a combination of permissions, and a specific subset of system calls' occurrence. An approach based on permissions' combination is studied in [29]. This scheme uses permission information present in the manifest of an Android application. These permissions' combinations are frequently requested by malicious applications and rarely requested by benign applications. Based on the permissions combinations, rules are generated in order to make classification of applications as benign or malicious. This model is then used for the classification of unknown applications.

An Android applications classification technique [30] and others [31] use bytecode for detection purposes. The bytecode of an application contains the accurate behavior of an application, which can provide enough information about the application's intentions. Similarly, malicious applications tend to have the same pattern of bytecodes which makes them unique and differentiates them from benign applications.

Android file's opcodes are considered the source of information for malware detection. A technique that uses opcodes as features for the identification of malware applications is studied in [32]. This method is based on state-of-the-art classifiers applied to the frequencies of opcode n-grams. Similarly, a very popular study [33] presents a detection mechanism based on features, such as sequences of opcodes combined with machine learning algorithms. As an initial input feature, it collects all the possible *k*-grams in a given set of applications. To determine key relevant features, a selection algorithm is applied and a classifier is trained based upon selected features. Another opcode-related study [34] applies static analysis over opcode distribution. In this particular experiment, Android executables are disassembled statically, and the opcodes' frequency distribution is extracted. These distribution patterns are then compared with nonmalicious executables' distributions. It is noticed that there is a significant difference between these distributions. This feature can be effectively applied to differentiate malware and benign Android applications. The exploitation of opcodes for malware detection is currently a hot topic for Android malware detection problems.

*2.4. Representative Handcrafted Features for Malware Detection.* After the extensive study of both dynamic and static analysis of Android malware, it is concluded that dynamic and static analyses use a separate set of features for malware analysis. Features and their importance can be studied and justified using domain knowledge and expertise from the computer security domain. Each approach considers different types of features and tries to justify their importance and relevance to the problem; there is no standard solution or set of features. The different categories of the features are classified in Table 1.

*2.5. Machine Learning Algorithms for Malware Detection.* Once the feature engineering and extraction step is performed, the next step is usually to apply machine learning algorithms, such as random forest, decision tree, SVM, Bayesian classifiers, KNN, and *k*-means clustering, for malware detection and classification. Each algorithm deals with the extracted features differently and tries to learn useful patterns. The parameters of algorithms are tuned considering the nature of features. Studies such as [35] and [36] demonstrate the use of machine learning algorithms for automatic Android malware detection.

$$y_k = \sum_i \left(w_{ki}.x_i\right), \tag{1}$$

$$E_n = \frac{1}{2}\sum_k \left(y_{nk} - t_{nk}\right)^2 \text{where; } y_{nk} = y_k\left(x_n, w\right), \tag{2}$$

$$\frac{\partial E_n}{\partial w_{ji}} = \left(y_{nj} - t_{nj}\right)x_{ni}. \tag{3}$$

*2.6. Deep Neural Networks and Deep Learning.* Deep learning or end-to-end learning is a paradigm of machine learning but is very practical [37]. It achieves great flexibility and power by learning to express the world in the form of a nested hierarchy of concepts, such that each complex concept is defined in terms of very simpler concepts. Concepts that are more abstract are defined in relation to less abstract concepts. This concept of end-to-end learning is not new but was there for decades. But recently with very strong hype, end-to-end learning is getting extraordinary attention. The more prominent contribution of end-to-end learning is automatic features engineering [38–41]. This property makes it unique and more powerful than all other existing machine learning techniques.

Table 1: Features set for Android malware analysis.

| | |
|---|---|
| Features for dynamic Android malware analysis | Behavioral footprinting<br>Dalvik instruction traces<br>System calls<br>API-level activity and information leakage<br>Heuristic-based filtering<br>Sensitive data handling<br>Run time behavior |
| Features for static Android malware analysis | Behavioral graph<br>API-level features<br>Weighted contextual API dependency graph<br>Control flow transition<br>Permissions<br>Opcode patterns<br>Opcode frequency histogram<br>Manifest and creator information<br>Information flow<br>Context, time, and connection-based network features<br>Statistical features<br>N-gram sequential features<br>Weighted sum of permissions<br>Internet component call graph<br>Signature-based features |

Convolutional Neural Network (CNN) is an example of a deep neural network; it tries to learn low-level concepts or features such as edges and may be lines in starting layers, then parts and pieces of objects, and then the high-level representation of the objects [42]. The general problem-solving approach of deep learning is different than conventional machine learning algorithms. These algorithms first divide the problem into subproblems, and then after solving all the subproblems, the results are compiled in a collective form. This is not the case in deep learning because it is end-to-end learning; i.e., just input the raw data and the classification results are collected as output. CNNs have multiple hidden layers where convolution operation is performed, a few dense layers near the end of the network, and an output and input layer. The input data is fed to the input layer along with random initial weights. The output is received from the output layer. All the calculations are performed in hidden layers. Equation (2) is the error function known as mean squared error, where $y$ is the predicted output and $t$ is the target output or may be named as a label. We can also consider $y$ as linear combinations of the given inputs as presented in equation (1), where $y_k$ is the output, $w_{ki}$ is the weight, and $x_i$ is the input. After prediction, the error is calculated using equation (2), $y_{nk}$ is the predicted output while $t_{nk}$ is the ground truth. This error function is for some input $n$, and all possible outputs or labels $k$. The term $x$ is a particular input having weight $w$. The calculated error is then backpropagated to hidden layers and weights are updated accordingly. This process is continued till the weights are optimized enough, and the value of the error function is converged. Equation (3) is the error function gradient considering some weight $w_{ji}$. This gradient is used in the backpropagation process.

*2.7. Deep Learning-Based Analysis for Android Malware.* There are a number of deep learning-based methods for Android malware detection. One of the methods is DroidSec [13]. It uses 200 different dynamic and static features with deep neural networks for malware detection. DeepDetector [14] is another approach, where eight different sets of features are considered to be used with deep neural networks. A multimodal deep learning method is proposed in [15]. The technique uses opcode features and method API features with deep neural networks. Another approach [16] extracts a sequence of API method calls and manually categorizes the dangerous APIs. The categorized API sequences are used to train deep neural networks. In [43], the authors use five different sets of features to perform classification using deep learning. Droiddetector [35] is considering three sets of features, i.e., required permission, sensitive API, and dynamic behavior, with deep neural networks.

*2.8. Motivation for the Proposed Approach.* Deep learning methods are popular because of deep feature extraction. All the existing deep learning-based techniques discussed in Section 2.7 have used handcrafted features for automated Android malware detection as depicted in Figure 2. These techniques do not exploit the deep feature extraction nature of deep learning methods but rather use engineered handcrafted features. To hand-design, an effective feature is a lengthy process. This approach is very costly and lengthy and requires intensive domain knowledge and expertise as already discussed in Section 1. There is a need for making this feature extraction process automatic and reduce human experts' intervention. Aiming at new applications, deep learning enables the acquisition of new effective feature representations from the available dataset for training. The major difference between deep learning and conventional methods is that deep learning automatically learns features from big data, instead of adopting handcrafted features as shown in Figure 3, which mainly depends on prior

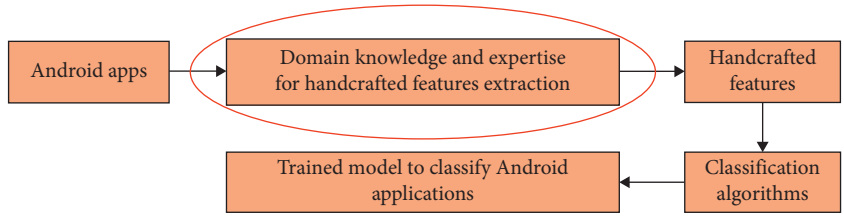


Figure 2: Conventional model for Android app classification.

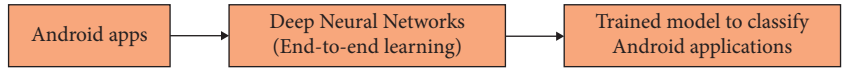


Figure 3: Proposed classification pipeline.

knowledge of designers and is highly impossible to take the advantage of big data.

Recent studies, such as [32, 33] and [34], offer solid justification that opcodes distribution and sequences provide important information to differentiate a malware from a trusted Android application. The problem is how to feed the opcodes information to deep neural networks as deep neural networks only work with numerical data. Another problem for Android application classification and malware detection is that there are only a limited number of datasets. The latest publicly available dataset is Drebin [26]. Drebin and all other existing datasets have Android apks structured directories and have such a format that they cannot be used to perform automatic malware detection using deep learning models [44].

First, we need to develop an opcode embedding technique to feed the opcodes to deep neural networks. One solution is one-hot encoding [45–47]. But this solution has shortcomings; i.e., one-hot encoding leads to inefficiency for high-dimensional input data and it does not capture the semantic relationship. There are extensive studies [48–49] that strengthen the concept that the vector embedding technique outperforms the conventional one-hot encoding solution. This motivates us to develop a vector embedding technique for opcodes which we name "Op2Vec." And secondly, based on the learned vector embeddings, we develop a dataset that will be used for the end-to-end detection of Android malware.

*2.9. Word Embeddings.* Word embedding is a technique used for embedding words into vectors. It does this task in such a fashion that the syntactic and semantic relationships between words are always preserved. Word2Vec is one of the very important subtasks for most of the applications of natural language processing (NLP). Word embedding is the collective name for a set of feature learning and language modeling techniques in the field of NLP where words and phrases from the available vocabulary are mapped to vectors of real numbers. Our work is inspired by TWEET2VEC [46], ATTACK2VEC [50], and ASM2VEC [51] where the authors have applied a word embedding technique to encode tweets, attacks, and assembly language functions into vectors. Word embedding is considered a very recent version of these embeddings which are relatively dense and have low dimensionality [52], which is very efficient in terms of computation. Get motivated from all these studies we are applying the Word2Vec to opcodes which we name Op2Vec, to learn vectors for opcodes. There are two common word embedding models, i.e., continuous bag-of-words (CBOW) and the skip-gram model. Because of the two severe limitations of CBOW [53], i.e., ignoring the order of words and ignoring the semantics of words, we will be considering the skip-gram model for our solution.

## 3. Proposed Methodology

In this section, we present details of the Op2Vec learning process and the designed dataset. The dataset consists of opcode sequences of Android applications. Initially, the dataset contains 28,570 Android applications. More applications can be easily added in the future to confirm the robustness of the proposed technique. The dataset design process consists of five phases. A pictorial view of all the steps involved in the proposed design process is shown in Figure 4. After the collection of Android apks from different online Android Play Stores, the first phase is the extraction of Dalvik Executable (.dex) files from apks. The second phase is the extraction of instructions from executable files. All the instructions are processed and only opcode sequences are extracted. In the third phase, opcode sequences of different files are combined to learn Op2Vec, i.e., opcode embeddings. In the very last stage, for every single file, opcodes are replaced by their corresponding vectors, learned in phase four to finalize the dataset. The final step is to feed the dataset to deep neural networks to ensure its validation for end-to-end learning.

*3.1. Collection of Benign Applications.* Benign applications are those applications that are solely designed for harmless and smooth fulfillment of user requirements. The main objective of these applications is to meet the user requirements. Some Android benign applications are free and others are paid, available in online markets. Because of DRM restrictions, we have mostly collected free Android applications for our dataset. Around 16,240 free benign Android applications are downloaded from Amazon Appstore,

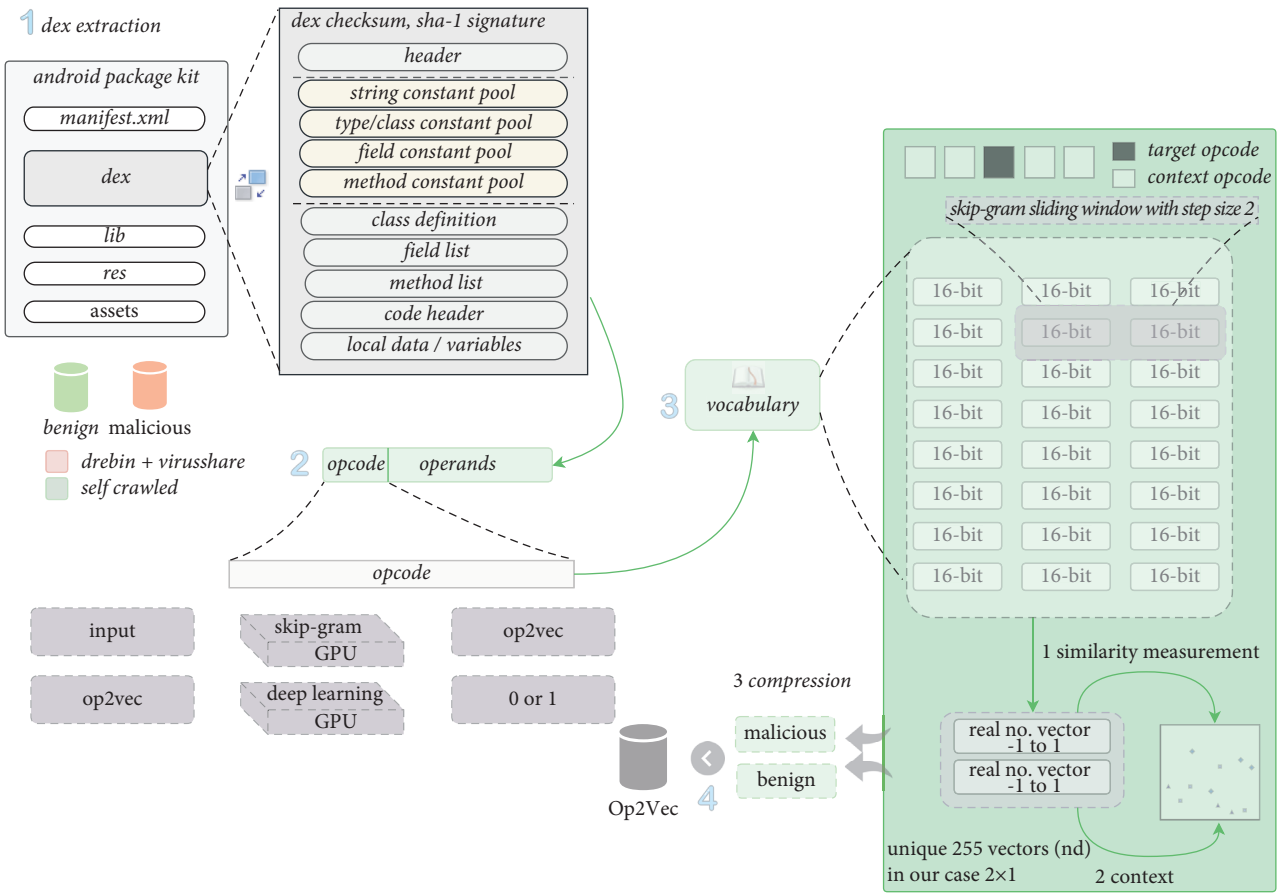


Figure 4: Stepwise workflow of Op2Vec model and dataset design.

SlideME, 1Mobile Market, and Google Play. A script (https://github.com/KaleemFAST/playstore-scraper-php.git) is designed to download applications from all the available sources.

*3.2. Collection of Malicious Applications.* Android applications are solely designed to meet the developer's interests at the cost of harming the application's users. A malicious payload is part of the application that carries malicious behavior. There are two main objectives [54] that mostly encourage the design of malicious applications: (1) without the user intention, triggering the malicious payload execution again and again for maximum benefit, and (2) escape from detection in order to have maximum life till the fulfillment of interests. We have collected around 12,330 malicious applications of different malware families, i.e., Androidbox, AnserverBot, and 12 other families.

*3.3. Dex Files Extraction.* The Android APK consists of files, classes.dex, resources.arsc, AndroidManifest.xml, subfolders, lib, assets, res, and META-INF. The file structure of the Android apk is depicted in Figure 4 step 1. All the files and folders have the necessary information regarding apk file. We are interested in the Dalvik Executable file, i.e., classes.dex. We extract this file from apk using a tool, named apktool (https://ibotpeaches.github.io). Apktool is a reverse engineering tool for Android apks. A script (https://github.com/KaleemFAST/Android%20End2End%20dataset%20design.git) is designed, and each apk file is unzipped through apktool. All the extracted classes.dex files are collected.

*3.4. Opcodes Extraction.* Dex file is the executable file for Android applications. The structure of the dex file is depicted in Figure 4 step 1. We have used a tool dedexer (https://sourceforge.net/projects/dedexer/), i.e., a disassembler tool for Android dex file. We get a dex.log file that produces all the necessary information of different portions of the dex file. Using the information given in the dex.log file, the desired code section of the dex file is filtered. From the code section of the dex file, all the opcode sequences are extracted to another file. This process is repeated for all the collected dex files. Now, we have files containing opcode sequences of dex files.

*3.5. Need for Opcode Embeddings.* Opcodes representation is in textual form, i.e., 05, *e*2, 03, 87, etc. There are two severe and main problems with this type of representation. (1) We can see the order relationship among these opcodes; i.e., 05 is greater than 03. In reality, there is no such relationship; i.e., opcodes are not comparable with each other and thus are not ordered [55]. If we feed this data to the deep neural network in this form, the network may consider this relation as a

feature (because of the feature engineering nature of deep neural networks), this reduces accuracy, and the learning process may be misled [56]. A technique is needed that should change this representation in such a fashion that assists the learning process and preserves the opcodes identity but still breaks the unnecessary ordering. (2) Deep neural networks do not work with textual/categorical data directly; rather, we have to change or encode its representation to numeric values.

The conventional encoding technique used is one-hot encoding [45–47]. This encoding technique has the following limitations:

(1) In some scenarios, one-hot encoding may be useful where the number of categorical variables is limited [45, 47]. But when the variables are not limited, it becomes very expensive and leads to inefficiency in most cases [49, 57]. In our case, we have 255 different opcodes so for each opcode $x_i$ if embedded into one-hot encoding, it will have size $x_i \in [0, 1]^{255\times 1}$, and for example, if a single file has 300000 opcodes, it will have 30000 vectors of dimensions $255 \times 1$. This data will exponentially grow if the number of files exceeds 50,000 figures, i.e., $50{,}000 \times 300{,}000$ vectors of dimensions $255 \times 1$. Training deep learning models on high-dimensional data having no spatial structure causes a major computational problem. It implies a network with an input layer of a very huge size, which greatly increases the number of weights, often making the training infeasible [58].

(2) It does not capture morphological resemblance between categories, and it also ignores the semantic relationship between the input categories [57]. This can be very useful for deep learning models to learn deep features from opcodes arrangement in Android source files [59].

$$p\left(w_{c,j} = w_{O,c} | w_I\right) = y_{c,j} = \frac{\exp\left(u_{c,j}\right)}{\sum_{j'=1}^{V} \exp\left(u'_j\right)}. \quad (4)$$

We definitely need to have an opcode embedding technique that overcomes all the above issues.

*3.6. Opcode Embeddings Using Skip-Gram Model.* From the previous section, it is clear that we need an opcode embedding technique. That is why we have introduced Op2Vec to get rid of all the listed issues. We have applied the skip-gram word embeddings technique for opcodes encoding. Skip-gram model [48] is a very prominent model in NLP that is used for Word2Vec. The words are embedded into vectors with the intuition that model needs to learn very similar and almost identical vectors for words having similar contexts. The complete architecture of the skip-gram is shown in Figure 5. Window size is selected based on the problem's nature. The input to the network is a one-hot vector that represents the input word and the output is also the number of one-hot vectors considering window size. While evaluating the trained network on a word given as input, the vectors that are obtained as output are probability distributions for nearby words, where from nearby words, we mean words lying inside the window selected for a particular vocabulary file given for training. The weights W in Figure 5, which are learned at the input layer, are the embedded representations of all the words in the vocabulary file. These probabilities are calculated using equation (4). In equation (4), $w_O$ and $w_I$ represent the output and input vectors, respectively. $V$ represents the length of the vector. $y$ is a training instance and $u$ is any given vector. In plain English, this equation states the prediction probability of a particular $j^{\text{th}}$ word of the $c^{\text{th}}$ panel, which equals $c^{\text{th}}$ output word, i.e., the actual value of the output vector index, conditioned on $w_I$. This equation decides the index value for a particular word in the output vector.

We have used the skip-gram to learn Op2Vec. Skip-gram uses word sequences, so for Op2Vec, words are analogous to opcodes. We applied this concept with the intuition that opcodes that appear in the same context must have similar vector representations. Word2Vec is an analogy to Op2Vec, i.e., learning Op2Vec, and encodes opcodes in such a manner that opcodes having similar semantics are assigned nearly identical vector representations.

## 4. Experimental Setup and Results

There are three main experiments that are carried out to justify the efficacy of the proposed approach. One is learning Op2Vec. The second one is the dataset development based on the learned Op2Vec, i.e., opcode embeddings for end-to-end learning of Android malware. And the third experiment is to feed the designed dataset to deep neural networks to validate the claim that the dataset can be used for the deep learning-based analysis of Android malware. In Figure 4, the gray boxes and cubes from left to right depict the Op2Vec learning process. The skip-gram model takes input and uses the GPU facility to learn Op2Vec. This Op2Vec will be used with deep learning models to perform end-to-end learning for Android malware detection and classify Android apps as benign 0 or malicious 1.

*4.1. Op2Vec: Learning Opcode Embeddings.* Words analogy to opcodes is considered in order to apply the word embeddings technique to opcode embeddings. The same steps and process of the skip-gram model, used for word embeddings in Section 3.6, are applied for opcodes. After the learning phase, opcodes are encoded into vector representations. This process consists of four subtasks which are listed as follows.

*4.1.1. Preprocessing Phase.* In the preprocessing phase, we consider 5,000 Android applications' dex files, 3,000 benign and 2,000 Malicious files out of the total 16,240 benign Android applications, and 12,330 malware files, respectively, for the development of vocabulary files to train our model. All the opcodes are collected into a single file. Now, this file is considered a vocabulary file for the learning phase. This

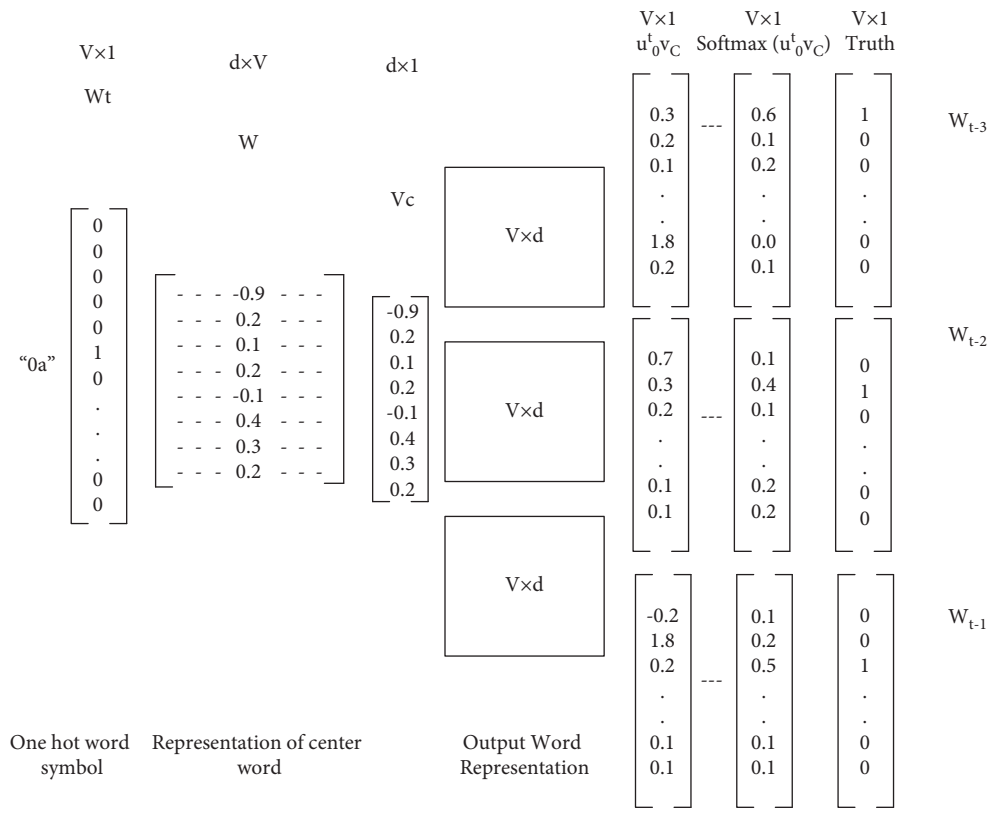


Figure 5: Skip-gram model.

vocabulary file has to be fed to the Op2Vec model for training. At the very start, we are not able to feed the opcodes directly to the neural network. The reason is that opcodes are represented in hexadecimal notations, i.e., strings; that is why we binarize this input with a one-hot encoding technique. As there are a total of 255 opcodes, so the length of the one-hot input vector is $255 \times 1$. For a particular opcode 0*a,* the one-hot vector is depicted in Figure 5. For 0*a,* the corresponding entry in the vector is fixed as 1 and all the other 254 entries are zeros. The output of this model network is a vector of the same size. It should also have 255 components. Every entry of the resultant output vector is the probability of an opcode selected randomly in the vicinity of the input opcode.

*4.1.2. Parameters Setting for Training Phase.* Op2Vec embeddings are learned using the skip-gram model. It is a neural network-based model having all the hidden, input, and output layers. All neurons in the hidden layers are without the activation functions, but all the neurons in the output layer use the softmax. Softmax is a type of regression used for multiclass classification. In softmax, for a given input $X$, a designed hypothesis tries to estimate $P(y = k|x)$, which is the probability for each value of $k = 1, \ldots, K$. This value of $k$ denotes the label of a particular class, so essentially the function gives us the probability of a particular input being in any class $k$. The network is trained on pairs of opcodes. The input vector is the one-hot representation of the input opcode. The output, which is also in the form of a one-hot vector, is all the opcodes inside the window except the input opcode, and we call it training output opcodes.

For computational simplicity and better visual representation, we trained our network for opcode vectors having two dimensions. We already know that the input vector size is 255 and we have selected the dimensions count as 2, so the representation of the hidden layer weight matrix is going to be in the form of a matrix with 255 columns and 2 rows. The ultimate goal of this setup is to learn the weight matrix. The output vectors are thrown out once we are done with learning. The network is trained to do the task, i.e., given any specific opcode in the middle of the opcodes sequence, and randomly pick one opcode from the vicinity, and the model tells us the probability for every opcode in the vocabulary to be that opcode we have selected randomly. The vicinity or nearby term is used because the skip-gram model uses the window as a parameter in its algorithm; typically, the window size parameter is set to 5 as recommended in the original documentation of the skip-gram. Window size 5 means 5 opcodes ahead and 5 opcodes behind the central opcode. We have hyperparameter, i.e., window size. For our problem, we fixed this hyperparameter to its default value as 5.

*4.1.3. Training Phase.* In the training phase of learning Op2Vec, the vocabulary file designed in Section 4.1.1 is used as input to the neural network. A sliding window of size 5 is adjusted to slide through all the opcodes in the vocabulary file till the end of the file. The neural network tries to

optimize its weight matrix after each iteration to adjust the probabilities at the output layer for the opcodes of the same context and semantics.

In Figure 6(a), a sliding window of size 2 is shown, i.e., two opcodes before the central opcode and two opcodes after the central one. For instance, 5 opcodes in a row are selected, and the opcode colored black is the input opcode for the network. After feeding this setup, the network tries to learn patterns in the form of statistical information of the number of occurrences of each pair, i.e., central opcode with any other opcode in the mentioned window. In this particular case where opcode *If-it* is the central opcode, the pairs are *(If-lt, If-le)*, *(If-lt, If-ge)*, *(If-lt, If-gt)*, and *(If-lt, If-eqz)*. Let us say the pair *(If-lt, If-ge)* occurred more frequently in the given vocabulary file, so when the learning phase is finished and we input the opcode *If-lt* to the network, it shows the high probability for the opcode *If-ge* in the output vector. The window is slid further to repeat this process for all opcodes in the vocabulary. This sliding procedure is depicted in Figures 6(b) and 6(c).

*4.1.4. Learning Op2Vec.* The input vector size is $255 \times 1$. The weight matrix for the first hidden layer is of size $2 \times 255$ as shown in Figure 5. The output of the hidden layer is a vector of $2 \times 1$. This vector is going to be input for the next layer, i.e., the output layer, where the weight matrices for this layer are of size $255 \times 2$. The output vectors of the network are of size $255 \times 1$, i.e., which is the probability distribution of all distinct 255 opcodes. The softmax is applied to the output of each neuron in order to get the values to sum up to 1. The columns of the learned weight matrix at the first hidden layer are the vector representations of all the 255 opcodes. After the network is trained, when we evaluate the network on a given input opcode, the output vector represents the probability distribution, i.e., a list of values in the form of floating points, not in the form of a one-hot vector that we obtain in the training phase.

*4.2. Op2Vec Results: Learned Opcode Embeddings.* After completion of the learning process, the weight matrix is divided into vectors to get the vector representation of all the 255 opcodes. The values range of all the vectors is in the interval $[-1, 1]$. For some very common opcodes, the learned vectors are listed in Table 2. These are two-dimensional vectors. It can be seen that the same categories of opcodes are represented by nearly identical vectors. Few of these vectors are plotted in Figure 7. Op2Vec has fixed all the issues of conventional one-hot encoding, discussed in Section 3.5, as follows:

(1) It is very clear from Figure 7 that the Op2Vec model learned vectors are depicted in such a manner that opcodes, having similar semantics, are represented by almost identical vectors. It is also depicted that opcodes, semantically different, are very apart from each other. If we look at Figure 7, we can clearly see that all the conditional statements *If-lt*, *If-le*, *If-eqz*, *If-ne*, *If-ge*, and *If-gt* are positioned very near in the space; this reveals that semantics are preserved in this sort of learning. Similarly, arithmetic opcodes *Mul-int*, *Sub-lg*, *Div-int*, *Div-lg*, and *Add-lg* are separately clustered. Because of the semantic similarity, they are almost identical. Same patterns can be observed for the rest of the opcode categories. So the results reveal the fact that Op2Vec has learned embeddings effectively and semantic relationships among opcodes are preserved. Intuitively, this is a very useful insight for deep learning models to learn deep features from opcodes arrangement in the Android source file. Introducing this relationship among opcodes will enhance the malware detection learning process [48, 49, 60], which is a contribution to automatic malware detection.

(2) The size of a single one-hot vector is $255 \times 1$. Table 2 shows that Op2Vec embeddings have reduced the $255 \times 1$ size of one-hot vectors to $2 \times 1$, which significantly decreases computationally complexity [58].

(3) Op2Vec embeddings have also fixed the limitation of ordering between originally extracted opcodes as there is no such real order in the generated vector representation.

*4.3. Development of the End-to-End Learning Dataset.* Now, when the Op2Vec embeddings are successfully learned, we can generate the proposed dataset that can be efficiently used for end-to-end learning analysis of Android malware. All the files generated in Section 3.4 are accessed one by one, and the opcodes are replaced by their corresponding vector representation learned in Section 4.2. Thus, the dataset is developed, and we claim that this is the first-ever attempt to develop a dataset that will be used for end-to-end detection of Android malware using deep neural networks.

*4.4. Feed the Developed Dataset to Deep Learning Models.* The claimed hypothesis that the dataset can be effectively used for end-to-end learning is validated by feeding the dataset to the CNN, i.e., a deep neural network/end-to-end learning model. Each file in the dataset has two-dimensional vectors corresponding to each opcode in the original dex file. So each file has two columns and a number of rows. For the network, we consider our input frame consists of two channels as each vector is of two dimensional. Each benign file is assigned a label as 0 whereas each malicious file is labeled as 1. The network's setup is all set to process the available files for end-to-end learning.

## 5. Comparison with Existing Datasets

This section draws a comparison among our designed Op2Vec dataset and six other very popular datasets in the malware analysis literature. The comparison is based on the following two parameters.

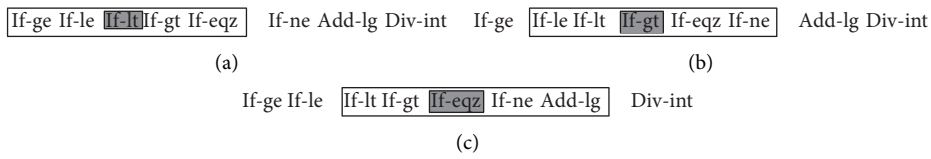


Figure 6: Sliding window over vocabulary opcodes.

Table 2: Learned Op2Vec: the learned vectors.

| Opcode | *X*-axis value | *Y*-axis value |
|---|---|---|
| If-ne | −0.2729177368 | −0.0875072266 |
| If-lt | −0.3726633597 | −0.017922292 |
| If-ge | −0.6149268202 | −0.0044448727 |
| If-gt | −0.6818177649 | −0.3873034379 |
| If-le | −0.3076827262 | 0.1643184456 |
| If-eqz | −0.2591792741 | 0.2236180313 |
| Mul-int | 0.2114985694 | 0.3691054416 |
| Sub-lg | 0.1262099695 | 0.1640332061 |
| Div-int | 0.1447551711 | −0.0522292025 |
| Add-lg | −0.0497673974 | −0.2025787514 |
| Div-lg | 0.1181362618 | −0.0835916175 |
| Iput-wide | 0.3703214875 | −0.0470563225 |
| Iput-byte | 0.3919335594 | −0.0879191859 |
| Iput-char | 0.462135628 | −0.3112477693 |
| Invoke-static | −0.278561079 | −0.3259538566 |
| Invoke-super | −0.6331899455 | −0.4750899485 |
| Invoke-virtual | −0.5944988213 | −0.5101055075 |
| New-array | 0.0389557098 | 0.6073178184 |
| Filled-new-array | −0.0966243253 | 0.5125404323 |
| New-instance | 0.0762253855 | 0.4074241374 |

*5.1. Features (Handcrafted or Deep Features).* One of the fundamental limitations of all other available datasets is handcrafted feature extraction which are employed for characterizing malware behavior. Our Op2Vec does not require handcrafted features. The datasets can be fed directly to the deep neural network for learning deep features.

*5.2. Feeding Information to the Deep Neural Network (One-Hot Encoding or Vector Embedding).* All the listed datasets and techniques have used one-hot encoding to feed the Android source code information to the classifiers. One-hot encoding has limitations that are discussed in Section 3.5. We have proposed Op2Vec which has fixed all the limitations and outperforms one-hot encoding as discussed in Section 4.2.

Drebin [26] has used a script for the automated extraction of different handcrafted features. The features are embedded in the one-hot encoding of the form $x_i \in [0, 1]^{545000\times1}$. SVM classifier is trained to classify applications based on their representative feature vectors. Another dataset that is used in [61]. A total of 42 handcrafted features of size $x_i \in [0, 1]^{42\times1}$ are extracted to use with LSTM. Similarly, the authors in [62] have used a dataset where 34,570 handcrafted features are extracted. This feature set is reduced to 413 using a feature selection technique. The input feature vector for the classifier is of size $x_i \in [0, 1]^{413\times1}$. Both [63, 65] have used 323 and 1,058 features, respectively, for machine

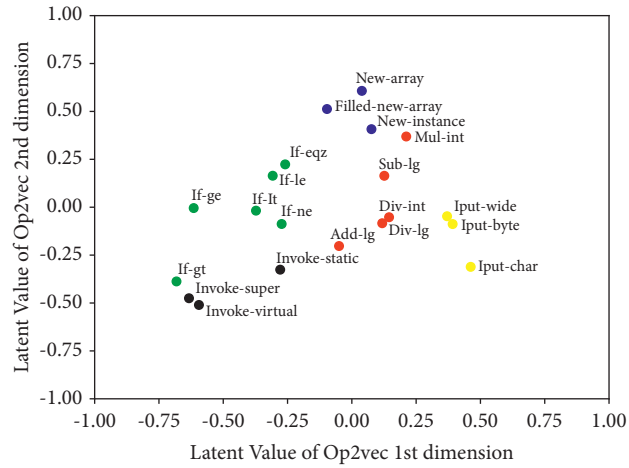


Figure 7: Learned opcode embeddings.

learning classifiers. Their input feature vectors are $x_i \in [0, 1]^{323\times1}$ and $x_i \in [0, 1]^{1058\times1}$, respectively. There are a number of datasets available for Android malware analysis and these datasets are only a collection of malicious Android applications. VirusShare [64] is one of the examples of such datasets. VirusShare has a total of 107,888 malicious Android applications. Different studies have used this dataset to learn insights into Android malware. A total of 482 features are considered and a feature vector of size $x_i \in [0, 1]^{482\times1}$ has been used to feed information to machine learning classifiers.

All the discussed datasets have a format that cannot be used to perform automatic malware detection using deep learning models. Most of these studies extract features from datasets, and these features are used with deep learning algorithms. Deep learning models have a very strong property, i.e., automatic deep feature extraction. They do not need handcrafted features; rather, raw data is sufficient for the training and learning process. Our proposed dataset is designed to exploit deep learning models for deep features. We have designed a dataset that can be fed directly to deep learning models. Unlike other datasets, no handcrafted features are required. The input encoded vector size is $x_i \in [-1, 1]^{2\times1}$, which has very few dimensions in comparison with other existing dataset techniques.

From Table 3, it is clear that in terms of features extraction and embedding technique our Op2Vec dataset is far better and adaptable as compared to the other datasets. Performing the Op2Vec type embedding technique can reduce dimensions up to $2\times1$. Adopting this dataset will allow learning deep features without extraction of

Table 3: Comparisons with existing datasets.

| Dataset | Files count | Malicious | Benign | Feature extraction | Features set | Feature vector |
|---|---|---|---|---|---|---|
| Drebin [26] | 129,013 | 123,453 | 5,560 | Handcrafted | 545,000 | $x_i \in [0, 1]^{545000\times1}$ |
| Vinayakumar et al. [61] | 2296 | 1,609 | 687 | Handcrafted | 42 | $x_i \in [0, 1]^{42\times1}$ |
| Wang et al. [62] | 23,000 | 13,000 | 10,000 | Handcrafted | 34,570 | $x_i \in [0, 1]^{413\times1}$ |
| Zhu et al. [63] | 11,000 | 8,000 | 3,000 | Handcrafted | 323 | $x_i \in [0, 1]^{323\times1}$ |
| VirusShare [64] | 107,888 | None | 107,888 | Handcrafted | 482 | $x_i \in [0, 1]^{482\times1}$ |
| Hou et al. [65] | 5,000 | 2,500 | 2,500 | Handcrafted | 1,058 | $x_i \in [0, 1]^{1058\times1}$ |
| AndroZoo [66] | 3,182,590 | 1,162,150 | 2,020,440 | Handcrafted | 50,000 | $x_i \in [0, 1]^{50000\times1}$ |
| Op2Vec dataset | 28,570 | 12,330 | 16,240 | Automated | Deep features | $x_i \in [-1, 1]^{2\times1}$ |

Table 4: Performance enhancement of existing opcode-based techniques.

| Reference (Year) | Features | Deep learning technique | Dataset | Reported results (Acc) (%) | Results with Op2Vec (acc) (%) |
|---|---|---|---|---|---|
| Parildi et al. (2021) [67] | Opcodes | VirusShare and native Win7 apps | RNN and LSTM | 95 | 96.83 |
| Ren et al. (2020) [68] | Opcodes | Google Play store and VirusShare | DNNs | 95.8 | 97.1 |
| Niu et al. (2020) [69] | Opcodes | VirusShare, Androzoo, and Pea Pods | LSTM | 97 | 98.77 |
| Pekta and Acarman (2020) [70] | Opcodes | Androzoo, Argus group, and GooglePlay | RNN and | 91.42 | 96 |
| Zhang et al. (2018) [71] | Opcodes | Microsoft in Kaggle 2015 and Benign apps | ResNet | 98.2 | 98.63 |
| McLaughlin et al. (2017) [72] | Opcodes | Genome project, McAfee Labs | CNN | 95 | 97.53 |

handcrafted features and with less computational complexity. The rest of Table 3 shows the files count in all the three datasets, malicious and benign files count, feature extraction method, and encoding techniques for features to be fed to machine learning and deep learning algorithms.

In order to demonstrate the significance of the proposed approach, some of the recent opcode-based deep learning techniques such as [67–72] are trained and tested with the Op2Vec dataset. For a fair comparison, the same experimental setup is used for the experiments with Op2Vec. It can be seen in Table 4 that the performance of the existing techniques significantly improves by incorporating Op2Vec embeddings. All the listed approaches achieve an average accuracy of 97.47%, where the highest accuracy is achieved with the setup suggested in [71]. It is evident from the results that Op2Vec which incorporates the semantic relationship of opcodes and deep features enhances the performance of deep learning techniques to detect Android malware.

## 6. Conclusion and Extensions

Previous work has shown that opcodes of executables have potential information. Opcodes can be considered as features in order to make discrimination between malware and benign Android applications. But these features are very hard to extract or notify. The handcrafted features or information extraction process is very expensive in terms of cost and time. In order to automate the process and effectively identify potential information and extract deep features, end-to-end learning is a perfect solution. This study concerns the learning of Op2Vec and the development of a novel dataset for end-to-end detection of Android malware. Op2Vec learning process employs a machine-learning algorithm to learn meaningful vector representations from opcodes of Android source files. The designed opcode embedding technique is used to develop a dataset for end-to-end detection of Android malware. The dataset will be used to learn useful patterns and information from the Android source code. We have not only developed the dataset but have also presented the design process and techniques involved in the dataset development. To the best of our knowledge, we believe this is the first state-of-the-art dataset for end-to-end Android malware detection. The product dataset of this research will be made openly available for further research concerning Android malware detection. Not only the dataset but also the designed process of the dataset will be made public so that in the future, new Android application files can be added to the dataset. This will make our technique robust to deal with newly emerging Android malware.

The proposed technique is one of the static Android malware analysis techniques. The limitation of this technique is that it may not capture the dynamic aspects of malware analysis. One of the future directions can be to combine the Dalvik instruction traces technique with the proposed approach to fix this limitation.

## Data Availability

The data used to support the findings of this study are available from (1) https://github.com/KaleemFAST/playstore-scraper-php.git, (2) https://ibotpeaches.github.io/Apktool/, (3) https://github.com/KaleemFAST/Android_End2End_dataset_design, and (4) https://sourceforge.net/projects/dedexer/.

## Conflicts of Interest

The authors declare that they have no conflicts of interest.

## References


[1] D. Geer, R. Bace, P. Gutmann et al., *Cyberinsecurity: The Cost of Monopoly*, Computer and Communications Industry Association (CCIA), Washington, D.C, USA, 2003.

[2] none, *Smartphone OS IDC. Market Share*, 2017.

[3] M. Aziz, A. Omar, and M. Mohaisen, "Amal: high-fidelity, behavior-based automated malware analysis and classification," *Computers & Security*, vol. 52, pp. 251–266, 2015.

[4] P. Faruki, A. Bharmal, V. Laxmi et al., "Android security: a survey of issues, malware penetration, and defenses," *IEEE communications surveys & tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[5] Q. Do, B. Martini, and K.-K. R. Choo, "Exfiltrating data from android devices," *Computers & Security*, vol. 48, pp. 74–91, 2015.

[6] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang, "Hybrid user-level sandboxing of third-party android apps," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, vol. 19–30, ACM, New York, NY, USA, April 2015.

[7] C. Lueg, *8,400 New Android Malware Samples Every Day*, 2017.

[8] C. Guo, J. Wang, and W. Zhu, "Smart-phone attacks and defenses," in *Hotnets III*, San Diego, CA, USA, 2004.

[9] J. Hamada, *New Android Threat Gives Phone a Root Canal*, 2011.

[10] A.-D. Schmidt, R. Bye, H.-G. Schmidt et al., "Static analysis of executables for collaborative malware detection on android," in *Proceedings of the in Communications ICC'09. IEEE International Conference on*, vol. 1–5, IEEE, Dresden, Germany, June 2009.

[11] T. Petsas, G. Voyatzis, A. Elias, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of android malware," in *Proceedings of the Seventh European Workshop on System Security*, vol. 5, ACM, Amsterdam, Netherlands, April, 2014.

[12] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and A. Sahin, "An android application sandbox system for suspicious software detection," in *Proceedings of the Malicious and unwanted software (MALWARE), 2010 5th international conference on*, pp. 55–62, IEEE, Nancy, France, October 2010.

[13] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-sec: deep learning in android malware detection," in *ACM SIGCOMM - Computer Communication Review*, vol. 44, pp. 371-372, ACM, 2014.

[14] D. Li, Z. Wang, and Y. Xue, "Deepdetector: android malware detection using deep neural network," in *Proceedings of the International Conference on Advances in Computing and Communication Engineering (ICACCE)*, pp. 184–188, IEEE, Paris, France, June 2018.

[15] T. Kim, B. Kang, M. Rho, S. Sezer, and E. Gyu, "A multimodal deep learning method for android malware detection using various features," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773–788, 2018.

[16] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "Android malware detection using deep learning on api method sequences," 2017, https://arxiv.org/abs/1712.08996.

[17] L.-K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *USENIX Security Symposium*, pp. 569–584, 2012.

[18] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pp. 281–294, UK, June 2012.

[19] Y. Zhou, Z. Wang, Z. Wu, and X. Jiang, "Hey, you, get off of my market: detecting malicious apps in official and alternative android markets," *NDSS*, vol. 25, pp. 50–52, 2012.

[20] M. Dimjasevic, S. Atzeni, I. Ugrina, and Z. Rakamaric, *Android Malware Detection Based on System Calls*, University of Utah, Salt Lake City, UT, USA, 2015.

[21] S. Kumar, G. Suarez-Tangil, S. Khan et al., "Droidscribe: classifying android malware based on runtime behavior," in *Proceedings of the Security and Privacy Workshops (SPW)*, pp. 252–261, IEEE, San Jose, CA, USA, May 2016.

[22] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: mining api-level features for robust malware detection in android," in *Proceedings of the International Conference on Security and Privacy in Communication Systems*, pp. 86–103, Springer, Sydney, NSW, Australia, September 2013.

[23] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: automated mining and characterization of fine-grained malicious behaviors in android applications," in *European Symposium on Research in Computer Security*, pp. 163–182, Springer, Berlin, Germany, 2014.

[24] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1105–1116, ACM, Scottsdale Arizona, USA, November 2014.

[25] A. Gorla, A. Zeller, V. Avdiienko, and K. Kuznetsov, "Mining apps for abnormal usage of sensitive data," in *Proceedings of the International Conference on Software Engineering*, Florence, Italy, May 2015.

[26] D. Arp, M. Spreitzenbarth, H. Gascon, K. Rieck, and C. E. R. T. Siemens, "Drebin: effective and explainable detection of android malware in your pocket," in *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[27] C. Liangboonprakong and O. Sornil, "Classification of malware families based on n-grams sequential pattern features," in *Proceedings of the Industrial Electronics and Applications (ICIEA), 2013 8th IEEE Conference on*, pp. 777–782, IEEE, Melbourne, VIC, Australia, June 2013.

[28] G. Canfora, F. Mercaldo, and C. A. Visaggio, "A classifier of malicious android applications," in *Proceedings of the Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pp. 607–614, IEEE, Regensburg, Germany, September 2013.

[29] S. Liang and X. Du, "Permission-combination-based scheme for android mobile malware detection," in *Proceedings of the Communications (ICC), 2014 IEEE International Conference on*, pp. 2301–2306, IEEE, Sydney, NSW, Australia, June 2014.

[30] B. Kang, B. Kang, J. Kim, and E. Gyu, "Android malware classification method: dalvik bytecode frequency analysis," in *Proceedings of the 2013 research in adaptive and convergent systems*, pp. 349-350, ACM, Montreal Quebec Canada, October 2013.

[31] A. Demontis, M. Melis, B. Biggio et al., "Yes, machine learning can be more secure! a case study on android malware detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, 2017.

[32] C. A. Visaggio, G. Canfora, and F. Mercaldo, "Mobile malware detection using op-code frequency histogram," in *Proceedings of the International joint Conference on e-business and Telecommunication*, Colmar, France, July 2015.

[33] Q. Jerome, A. Kevin, S. Radu, and T. Engel, "Using opcode-sequences to detect malicious android applications," in *Proceedings of the Communications (ICC), 2014 IEEE International Conference on*, pp. 914–919, IEEE, Sydney, NSW, Australia, June 2014.

[34] D. Bilar, "Opcodes as predictor for malware," *International Journal of Electronic Security and Digital Forensics*, vol. 1, no. 2, pp. 156–168, 2007.

[35] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.

[36] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "Stormdroid: a streaminglized machine learning-based system for detecting android malware," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pp. 377–388, ACM, May 2016.

[37] S. Jürgen, "Deep learning in neural networks: an overview," *Neural Networks*, vol. 61, pp. 85–117, 2015.

[38] S. Frank, G. Li, X. Chen, and Y. Dong, "Feature engineering in context-dependent deep neural networks for conversational speech transcription," in *Proceedings of the Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*, pp. 24–29, IEEE, Waikoloa, HI, USA, December 2011.

[39] M. Islam, K. N. Khan, and M. S. Khan, "Evaluation of preprocessing techniques for U-Net based automated liver segmentation," in *Proceedings of the 2021 International Conference on Artificial Intelligence (ICAI)*, pp. 187–192, Islamabad, Pakistan, April 2021.

[40] B. Ahmad, F. A. Khan, K. N. Khan, and M. S. Khan, "Automatic classification of heart sounds using long short-term memory," in *Proceedings of the 2021 15th International Conference on Open Source Systems and Technologies (ICOSST)*, pp. 1–6, Lahore, Pakistan, December 2021.

[41] R. Hasib, K. N. Khan, M. Yu, and M. S. Khan, "Vision-based human posture classification and fall detection using convolutional neural network," in *Proceedings of the 2021 International Conference on Artificial Intelligence (ICAI)*, pp. 74–79, Islamabad, Pakistan, April 2021.

[42] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 609–616, ACM, Montreal Quebec, Canada, June 2009.

[43] N. Abdelmonim and Y. Li, "Using deep neural network for android malware detection," 2019, https://arxiv.org/abs/1904.00736.

[44] X. Li, Y.-H. Lian, and Y. Hong, "Classification of mobile apps with combined information," in *Proceedings of the IEEE International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pp. 193–198, IEEE, Chengdu, China, July 2016.

[45] J. Gu, G. Wang, and T. Chen, "Recurrent highway networks with language cnn for image captioning," 2016, https://arxiv.org/abs/1612.07086.

[46] S. Vosoughi, P. Vijayaraghavan, and D. Roy, "Tweet2vec: learning tweet embeddings using character-level cnn-lstm encoder-decoder," in *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*, pp. 1041–1044, ACM, Pisa Italy, July 2016.

[47] A. C. H. Choong and N. K. Lee, "Evaluation of convolutionary neural networks modeling of dna sequences using ordinal versus one-hot encoding method," in *Proceedings of the International Conference on Computer and Drone Applications (IConDA)*, pp. 60–65, IEEE, Kuching, Malaysia, November 2017.

[48] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the Advances in neural information processing systems*, pp. 3111–3119, December 2013.

[49] J. Pennington, R. Socher, and C. D. Manning, "Glove: global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543, 2014.

[50] Y. Shen and G. Stringhini, "Attack2vec: leveraging temporal word embeddings to understand the evolution of cyberattacks," 2019, https://arxiv.org/abs/1905.12590.

[51] S. H. H. Ding, C. M. F. Benjamin, and P. Charland, "Asm2vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," IEEE, San Francisco, CA, USA, May 2019.

[52] L. K. Senel, I. Utlu, V. Yucesoy, A. Koc, and T. Cukur, "Semantic structure and interpretability of word embeddings," 2017, https://arxiv.org/abs/1711.00331.

[53] B. Wang, A. Wang, F. Chen, Y. Wang, and C.-C. J. Kuo, "Evaluating word embedding models: methods and experimental results," *APSIPA transactions on signal and information processing*, vol. 8, 2019.

[54] B. Bashari, M. Masrom, and S. Ibrahim, "Camouflage in malware: from encryption to metamorphism," *International Journal of Computer Science and Network Security*, vol. 12, no. 8, pp. 74–83, 2012.

[55] C. LeDoux and A. Lakhotia, "Malware and machine learning," in *Intelligent Methods for Cyber Warfare*, Springer, Berlin, Germany, 2015.

[56] A. Jović, K. Brkić, and N. Bogunović, "A review of feature selection methods with applications," in *Proceedings of the 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 1200–1205, IEEE, Opatija, Croatia, May 2015.

[57] P. Cerda, G. Varoquaux, and B. Kégl, "Similarity encoding for learning with dirty categorical variables," *Machine Learning*, vol. 107, no. 8-10, pp. 1477–1494, 2018.

[58] P. I. Wójcik and M. Kurdziel, "Training neural networks on high-dimensional data using random projection," *Pattern Analysis & Applications*, vol. 22, no. 3, pp. 1221–1231, 2019.

[59] Y. Yan, X.-C. Yin, B.-W. Zhang, C. Yang, and H. Hong-Wei, "Semantic indexing with deep learning: a case study," *Big Data Analytics*, vol. 1, no. 1, 2016.

[60] K. Yoon, "Convolutional neural networks for sentence classification," 2014, https://arxiv.org/abs/1408.5882.
[61] R. Vinayakumar, K. P. Soman, P. Poornachandran, and S. Kumar, "Detecting android malware using long short-term memory (lstm)," *Journal of Intelligent and Fuzzy Systems*, vol. 34, no. 3, pp. 1277–1288, 2018.
[62] W. Wang, M. Zhao, and J. Wang, "Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network," *Journal of Ambient Intelligence and Humanized Computing*, vol. 10, no. 8, pp. 3035–3043, 2019.
[63] D. Zhu, H. Jin, Y. Yang, D. Wu, and W. Chen, "Deepflow: deep learning-based malware detection by mining android application for abnormal usage of sensitive data," in *Proceedings of the IEEE symposium on computers and communications (ISCC)*, pp. 438–443, IEEE, Heraklion, July 2017.
[64] J.-M. Roberts, "Virus share," 2011, https://virusshare.com/.
[65] S. Hou, A. Saas, L. Chen, Y. Ye, and T. Bourlai, "Deep neural networks for automatic android malware detection," in *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pp. 803–810, Sydney, NSW, Australia, July 2017.
[66] A. Kevin, T. F. Bissyandé, J. Klein, and Y. L. T. Androzoo, "Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16*, pp. 468–471, ACM, New York, NY, USA, 2016.
[67] E. S. Parildi, D. Hatzinakos, and Y. Lawryshyn, "Deep learning-aided runtime opcode-based windows malware detection," *Neural Computing & Applications*, vol. 33, no. 18, Article ID 11963, 2021.
[68] Z. Ren, H. Wu, Q. Ning, I. Hussain, and B. Chen, "End-to-end malware detection for android iot devices using deep learning," *Ad Hoc Networks*, vol. 101, Article ID 102098, 2020.
[69] W. Niu, R. Cao, X. Zhang, K. Ding, K. Zhang, and T. Li, "Opcode-level function call graph based android malware classification using deep learning," *Sensors*, vol. 20, no. 13, p. 3645, 2020.
[70] A. Pektaş and T. Acarman, "Learning to detect android malware via opcode sequences," *Neurocomputing*, vol. 396, pp. 599–608, 2020.
[71] X. Zhang, M. Sun, J. Wang, and J. Wang, "Malware detection based on opcode sequence and resnet," in *Proceedings of the International Conference on Security with Intelligent Computing and Big-Data Services*, pp. 489–502, Springer, Guilin, China, December 2018.
[72] 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*, pp. 301–308, 2017.