Robust Deep Learning Early Alarm Prediction Model Based on the Behavioural Smell for Android Malware

Article *in* Computers & Security · February 2022

DOI: 10.1016/j.cose.2022.102670

2 authors:

Eslam Amer
Misr International University
47 PUBLICATIONS **415** CITATIONS
SEE PROFILE

Shaker El-Sappagh
Sungkyunkwan University
114 PUBLICATIONS **2,080** CITATIONS
SEE PROFILE

Robust Deep Learning Early Alarm Prediction Model Based on Behavioural Smell for Android Malware

Eslam Amer[a,*], Shaker El-Sappagh[b,c]

[a]Faculty of Computer Science - Misr International University - Cairo - Egypt
[b]Faculty of Computer Science and Engineering, Galala University, 435611, Suez, Egypt.
[c]Information Systems Department, Faculty of Computers and Artificial Intelligence, Benha University, 13518, Banha, Egypt

ARTICLE INFO

Keywords:
Android malware prediction
API calls
Contextual behaviour
System calls
Permissions
Behavioral analysis
Process mining
Sequence reformulation

ABSTRACT

Due to the widespread expansion of the Android malware industry, malicious Android processes mining became a necessity to understand their behavior. Nevertheless, due to the complexities of size, length, and associations of some essential and distinguishing Android applications' features such as API calls and system calls, mining the malicious Android processes become a prominent obstacle. The malicious process mining obstacle is also coupled with the increasing rate of zero-day attacks, with no prior knowledge about those kinds of behaviors. Hence, malware detection alone is no longer enough; instead, we need new methodologies to predict malicious behaviors early. In this paper, we proposed a behavioral Android malware smell predictor model. Our model relied on various static and dynamic features. We overcame the problem of massive feature size and complex associations by encapsulating related features in a few cluster classes. Accordingly, the cluster classes are exchangeably used to represent the features in the original calling sequences. Regarding substantially long sequences, experimental results showed that our model could predict whether a process is behaving maliciously or not based on rapid-sequence-snapshot analysis. The proposed model counted on the LSTM model to classify the reformed API and system call sequences snapshots. Moreover, we used ensemble machine learning classifiers to classify Android permissions. We trained the LSTM model using random snapshots of the newly formed API and system call cluster sequences. We tested our model against common ransomware attacks. We found that our trained LSTM model showed stable performance at a particular snapshot size. The model showed competitive accuracy in predicting unseen sequences. Accordingly, we proposed an early alarm solution for blocking malicious payloads instead of identifying them after their fulfillment. Hence, we can avoid the cost of future damage.

1. Introduction

Recently, the Android operating system has been widely used as the preferred environment for smartphones, tablets, and even the Internet of Things (IoT) devices [1]. In particular, the flexibility of smartphones, along with their high computing capabilities, has gained proper interest than personal computers. However, due to the massive evolution in the mobile world, most mobile applications have become the primary carrier of malware nowadays. The reason is that smartphones and mobile devices, generally, became the most preferred tools to access online resources [2]. Therefore, it becomes essential to find out new detection techniques that automatically identify malware for mobile devices, particularly those that employ the Android operating system, since it serves over 80% of the market share compared to iOS (roughly 15%) [3].

Cyber attackers are concentrating their endeavors in continuously exhibiting new and complicated techniques for impeding malware detectors [4]. According to statistical security reports[1], it stated that about 430,000 attacks were launched in May 2020, and that number is jumped by 3.6% in 30 days, increasing by 6.26% between July and August 2020. The report also showed that the U.S. was the worst influenced by mobile ransomware, estimating 63% of the world infections. As the long-term battle against malware progressed, the large number of hitherto unheard-of assaults and evasion strategies exposed an unassailable fact: keeping the detection engine up to date is entirely ineffective [5, 6].

*Corresponding author
ORCID(s):
[1]https://www.statista.com/statistics/680705/global-android-malware-volume/ [accessed online 5-December-2020]

Google tried to alleviate the spreading of malware by introducing Google Play Protect service[2]. Google Play Protect automatically examines applications even after installation to guarantee that the installed applications remain safe 24/7. However, most third-party stores cannot cope with scanning and detecting submitted harmful applications. For example, Google Play Protect showed deficiencies when tested against malware discovered from 90 days in 2017 [7]. Consequently, there is a necessity for efficient supplementary methods to detect zero-day Android malware to subdue the challenges above.

Current Android malware analysis approaches are categorized into static analysis, and dynamic analysis [8, 9]. Static analysis involves reverse-engineering an application code to discover privacy leakage and vulnerabilities in Android apps or to obtain different files included in the package (i.e., the Android Manifest) [10, 11]. However, static analysis methods are vulnerable to code polymorphism, and malware obfuscation [12]. Conversely, dynamic analysis methods attempt to run an application in a secured virtual environment to track and access sensitive information during runtime [12, 13]. The dynamic or behavioral analysis is used to examine an application's behavior while it is running: it analyses all requests for files and data, processes, and connections. That type of analysis recognizes all suspicious actions, allowing us to determine in advance if a file is harmful before it is released into the network and executes a malicious activity. Accordingly, the Android intrusion detection system analyzes lists of processes, system calls, network traffic, and other factors that enable the identification of intrusions. Dynamic analysis techniques are effective [14, 15]; however, they require a considerable amount of time to analyze an application behavior. Therefore, implementing dynamic analysis methods on resource-constrained smart devices remains challenging.

An essential part of understanding how Android processes behave is being able to characterize each one precisely and figure out what the process's goal is. Android apps often use APIs, permissions, and system calls to communicate with hardware to carry out their intended functions. Therefore, these features can provide an insight into the behavior of applications [16, 13]. The hidden intent, which lies in the API calls, system calls, and permissions list sequence structure, can reveal the *smell* of malicious versus non-malicious activities. Therefore, we need to understand the distinction in the *semantic* interaction between APIs and system calls along with permissions list in both Android malware and goodware Apps. Accordingly, we can capture the *distinctive behavioral smell* between both malicious and non-malicious activities.

In this paper, we introduced the *malicious Android smell predictor*, which is an extension of our previous work [17] for analyzing Android malware and goodware applications. Our proposed model relied on hybrid static and dynamic analysis features extracted from Android apps. In our model, we used the extracted API calls, system calls, and permission sequences from executable Android apps to create an *early* behavioral smell prediction model. Our prediction model captured the implicit *contextual* relations that emerged through the aforementioned features' interactions while the application was functioning. We tested our model against new unseen data which contained goodware and malicious ransomware samples, and it proved its effectiveness in identifying the category of unknown samples. The main contributions of this paper can be summarized as follows:

- Introducing an early malicious Android smell predictor that contextually analyzes and predicts malicious Android applications.
- Performing a hybrid analysis model through static and dynamic features.
- Highlighting the outperformance of the proposed Android smell prediction approach compared to various state-of-the-art detection methods.

The rest of the paper is organized as follows: the related works and background are discussed in Section 2. Section 3 presents our proposed malware detection model. Datasets and the empirical evaluation of the proposed work are given in Section 4. Section 5 briefly discusses the significant challenges to current malware analysis tools. Finally, Section 6 concludes our paper.

2. Related Work

The widespread usage of Android applications has increased user concerns about potentially unwanted applications. Therefore, Android users are usually looking for robust tools that detect and analyze malicious Android applications. In this section, we discuss the most relevant research that utilized API calls, system calls, and permissions for Android malware analysis and detection.

[2]https://www.android.com/play-protect/

Table 1
Top-10 ranked permissions used by malicious Android apps

Rank	Permission
1	android.permission.INTERNET
2	android.permission.READ_PHONE_STATE
3	android.permission.ACCESS_NETWORK_STATE
4	android.permission.WRITE_EXTERNAL_STORAGE
5	android.permission.ACCESS_WIFI_STATE
6	android.permission.READ_SMS
7	android.permission.WRITE_SMS
8	android.permission.RECEIVE_BOOT_COMPLETED
9	android.permission.ACCESS_COARSE_LOCATION
10	android.permission.CHANGE_WIFI_STATE

Table 2
Top-10 ranked permissions used by goodware Android apps

Rank	Permission
1	android.permission.INTERNET
2	android.permission.WRITE_EXTERNAL_STORAGE
3	android.permission.ACCESS_NETWORK_STATE
4	android.permission.WAKE_LOCK
5	android.permission.RECEIVE_BOOT_COMPLETED
6	android.permission.ACCESS_WIFI_STATE
7	android.permission.READ_PHONE_STATE
8	android.permission.VIBRATE
9	android.permission.ACCESS_FINE_LOCATION
10	android.permission.READ_EXTERNAL_STORAGE

2.1. Android Malware Detection Based on Permissions

Permissions' analysis is one of the most widely utilized feature in the static analysis of Android applications. Consequently, they are an integral part of Android's security architecture. The Android system contains a permission-based security mechanism to prevent applications from exposing users' private information. Therefore, a set of permissions are requested from the user, who has to accept them to allow the installation of any application [18, 19]. When permissions are approved by the user, the application starts the installation process on his device. The Android system permissions have several forms. Nevertheless, the sequence of permissions may expose certain potentially dangerous behaviors [19]. For instance, if an application requests permission for network connection as well as SMS accessibility permissions, then the application may steal users' SMS forms and then disseminate them over the Internet. To correctly identify Android malware, research work [20, 21, 22] have directed their attention to permissions rather than other static features. Permissions are viewed as the initial protection line against an attacker's malicious intentions.

Although permissions are easily extractable from the Android Manifest (Androidmanifest.xml), research work [3, 22] empirically shows that there is a great overlap in permissions between malicious and non-malicious applications. For example, Table 1 and Table 2 showed the top-10 permissions used in malware and goodware Android samples, respectively. Permission overlap between malware and goodware applications is predictable since practically all modern applications rely heavily on those common permissions. Consequently, the detection accuracy for approaches that purely relied on permissions as a distinctive feature achieved around (90%-93%) [23, 22].

2.2. Android Malware Detection Based on API calls

API calls are one of the terms used to describe the interaction between applications and the Android platform. Consequently, API calls are mainly required by every Android application to communicate with the device [24]. As a consequence, much research work has focused on API calls as a means of detecting malicious applications [25]. Accordingly, it is critical to track all API calls and their dependencies as they occur. These types of information may be obtained using both static and dynamic analysis methods.

Malicious applications usually hide or alter the API sequence through code obfuscation to avoid detection. Therefore, much of the research work emphasized the significance of extracting API calls during application execution, Where API calling sequences are considered a distinct feature for detecting malicious applications [26, 27, 28, 13]. Consequently, the API dependency calling graph is viewed as a significant feature that can describe the relations between APIs in malicious or non-malicious applications [13, 15, 17].

Zhang et al.[29] relied on the association rules to study the semantics between different API calls. The authors achieved an accuracy detection of 96% on the Driben dataset, which contains 5.9K, 5.6K goodware, and malware applications, respectively. Qiao et al.[30] introduced a malware detection approach relying on the API calls and permissions indexed in the Android applications. They experimented with the use of Support Vector Machines (SVMs), Random Forest, and Artificial Neural Networks over a dataset containing 6260 applications. They showed that their API calls detection approach outperforms the permission-based detection one. The API calls detection methods achieved an accuracy of (81.68% – 94.41%) concerning the machine-learning algorithm used. Chao et al.[31] proposed a DroidMiner to identify and categorize malicious applications into families based on the API calls. DroidMiner was evaluated using a collection of 10,403 goodware applications and 2466 malicious applications that belong to 68 malware families. The authors experimented with their methodology using four machine learning algorithms: naïve Bayes, decision trees, support vector machines (SVM), and random forest (RF). DroidMiner achieved accuracy in the range of (82.2% - 95.3%).

Onwuzurike et al.[32] proposed a behavioral model called MAMADROID. They relied on the Markov chain to model the abstracted API call implemented by an application to extract behavioral classification features. The authors evaluated MAMADROID over a dataset of 35.5K malicious and 8.5K goodware applications where it achieved a detection accuracy of 99%. Taheri et al. [33] relied on the extracted APIs and permission features to implement four different malicious detection approaches. They used hamming distance to determine the similarity between different samples. In their work, they applied first nearest neighbors (FNN), weighted all-nearest neighbors (WANN), all nearest neighbors (ANN), and k-medoid-based nearest neighbors (KMNN). They achieved an accuracy of 0.96-0.98 using permission features, whereas using API calls, they reached an accuracy of 0.99.

2.3. Android Malware Detection Based on System calls

System calls are considered the most significant feature in dynamic analysis. The reason for its prominence is that it acts as an interface between the applications and the Android OS. User or application processes cannot directly access system resources [34, 35, 36]. Accordingly, system calls provide applications with useful functions to perform activities related to networking, files, and a variety of other tasks. Therefore, we may deduce the behavior of malicious apps by studying the sequence of system calls.

Yu et al. [37] used permissions and system calls to characterize the Android applications' malicious behaviors. They relied on the recurrent neural network (RNN) and the feedforward neural network (FNN). Xioa et al. [38] and Dimja et al. [39] introduced the BMSCS and MALINE models, respectively. They used the n-gram models to find featured subsequences from system call sequences. Furthermore, they employed ML techniques to reveal malicious Android applications. MALINE took into account the reciprocal distance between each pair of system calls as a distinguishing feature in both malicious and non-malicious sequences. However, BMSCS computed the transition probability between each pair of system calls as a discriminatory feature between malicious and non-malicious sequences.

Canfora et al. [40] considered the co-occurrence dependency ratio between system calls as a weighted feature in the whole sequence. Saracino et al. [41] used different ML techniques to characterize malware over constructed a system calls feature vectors. In their work, the feature vector contains only eleven system calls, which is inadequate and also excludes the interrelationships between system calls. Xiao et al.[42] used the Long Short-Term Memory model (LSTM) to propose a malicious system call detection method. In their work, the authors aimed to distinguish malicious call sequences from goodware ones. Therefore, they trained two classifiers for goodware and malware using sequences of system calls for applications related to each category. Through LSTM models, the classifiers will be provided with the sequence history up to a particular system call. Accordingly, the LSTM models will classify any application based on the highest probability of a sequence in both goodware and malicious models. The authors showed that their approach achieved an accuracy rate of 93.7%.

2.4. Limitations

Previous research efforts endeavored to apply machine and deep learning models to identify the relationships between different Android features [43]. Consequently, they could be able to model the variations and differences in the

behavioral patterns between malicious and goodware applications. Machine learning algorithms relied on pre-selected feature occurrence patterns such as APIs, system calls, and permission lists [44, 45, 46]. However, these approaches were manipulated through experts for feature engineering, which means they are likely to change. Moreover, with any future change to feature patterns, the detection approaches may falsely classify the new patterns. Therefore, the ML detection approaches became more vulnerable to change [47, 48, 49].

Additionally, there was a difficulty with the use of insufficiently large or imbalanced datasets, which created some doubt regarding the true success of various procedures in the real environmental situations. None of these approaches have been rigorously evaluated in what is likely the most challenging scenario (e.g., a proper zero-day situation), with the exception of [50], which attempted a zero-day scenario. Moreover, in some classes of Android malware, namely Adware, there are minor variations between goodware and malicious applications [51]. Consequently, it's critical to build detection mechanisms that are specially designed to identify specific sorts of malware.

The research literature on malware analysis emphasized the significance of API calls or system calls to understand the malicious process behavior [52, 13, 15]. Although the API/system calling sequences could provide an apparent description regarding the behavior of an application, it was stated by [49] that there are no benchmark datasets that contain tens of thousands of goodware and malicious applications. Perhaps one of the common reasons is the time required to execute and monitor every application. Moreover, intelligent malware authors usually employ anti-emulation or anti-sandbox techniques to hide the malicious core of malware when virtual environment instrumentation is detected [53].

Deep learning models, especially LSTM, have proved their effectiveness in learning features from time-series data. However, due to the extreme length sequences, as in the case of API or system calls sequences, LSTM showed to be ineffective when learned from long sequences [54, 55]. The reason behind that is that the LSTM consumes a considerable amount of time in calculating errors by the backpropagation algorithm.

Accordingly, in our paper, we proposed a malware prediction model that can predict whether an application can lead to malicious behavior or not based on rapid-sequence-snapshot analysis. Our model also introduced a solution that alleviates the problem of LSTM long sequences by relying on small sequences. We experimentally showed that our model efficiency is not affected when the sequence length increases.

3. Proposed model

According to the information security aspect, Android malware detection techniques that study malware behavior mainly rely on dynamic analysis features like API or system call sequences. Current research work usually tries to detect suspicious sub-sequence n-gram patterns that frequently occur in API or system call sequences. That's usually what malware developers think, too. Therefore, from the viewpoint of malware developers, we are tracking their traits. As a result, malware developers will have the lead every time. Responsibly, we set out to thoroughly comprehend malware and goodware behaviors. Although both categories use the same behavioral features like APIs, system calls, and other static features such as permissions, various associations of the same feature could yield malicious or normal activities. Accordingly, there must be some distinctions that distinguish between the two activities. Upon capturing these distinctions, it is possible to *predict* whether incoming new sequences are malicious or not.

Feature engineering plays a vital role in any learning-based app classification model. More profound knowledge of the proliferation of malicious and normal applications would result in the development of effective long-term security solutions for Android. For example, the study of run-time behavioral differences between malware and goodware apps may reveal behavioral characteristics that consistently separate these two groups over time. The behavioral metrics of these characteristics can then be used as distinctive features for training effective classification models for deploying sustainable, dynamic malware detectors or predictors.

According to [56, 57, 27, 13], behavioural features can depict the conceptual program execution. Thus, they are known to be one of the most informative approaches that describe the process behavior. In this paper, we identified and determined the semantics behind the processes through a hybrid static and dynamic features. Eventually, we proposed our Android smell predictor model outlined in Fig. 1. Based on the rapid-sequence-snapshot analysis, we showed that our model could predict whether a given sequence is malicious or not. The proposed Android malicious smell predictor model has three phases, namely, behavioral feature extraction, feature reformulation, and classification. We will explain each phase in detail in the following subsections.

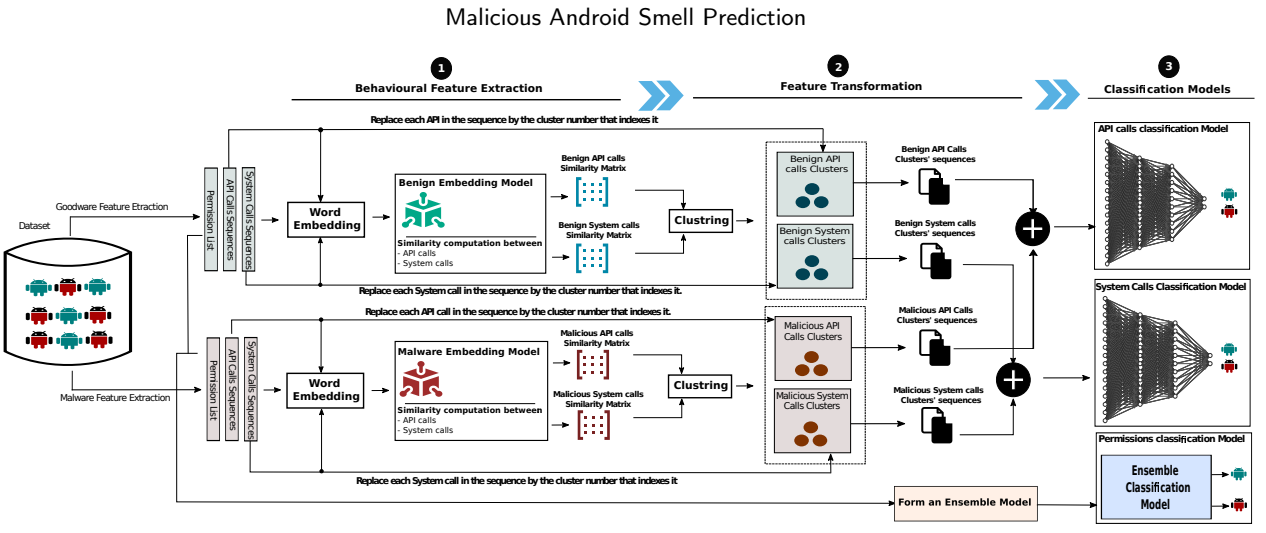

Fig. 1: Android Malicious Smell Predictor Model

3.1. Behavioral feature extraction

The main objective behind that phase is to extract the behavioral distinctions between malicious and non-malicious applications. We stated before that both malware and normal applications use the same features such as APIs, system calls, and permissions to perform their activities. However, the behavior intention behind the feature calling chain in a given sequence is considered the main difference between malicious and non-malicious behaviours. Therefore, we asserts in our previous studies [13, 15, 17] that there is no arbitrary ordering in the chains of API or system functions in the malware sequences.

Undoubtedly, specific semantic conceptual patterns that accomplish malicious behaviors are implicitly encoded within the feature calling chain. These semantic conceptual patterns are remarkably consistent across various malicious applications. By extracting those patterns from a significant number of malware sequences, we may be capable of identifying the implicit contextual interrelationships between the individual calls in those sequences.

Initially, our model started by feature extraction from our malware and goodware training dataset. As shown in Fig. 1, the outcomes are a set of permissions, API calls, and system calls features for both malware and goodware applications. Fig. 2 and Fig. 3 represent sample permission list and system calls that are extracted from an Android application.

In our model, we used word embedding to derive the contextual nature of the relationships within API and system calls in malware and goodware calling sequences. We relied on Word2Vec, which is considered one of the most commonly used word embedding models. As described in Fig. 1, we fed the word embedding models with a corpus of malware and goodware API and system calls sequences. The production of word embedding as outlined in Fig. 1 is two embedding models, namely goodware and malware embedding models. In our experiments, we parameterized our Word2Vec model with dimensional feature vectors size of 300, a window size of 6, workers of 6.

As shown in Fig. 1, we used the resultant embedding models to compute the contextual similarity between individual API and system functions. Word2Vec model provides functions such as the $model.similarity(A, B)$ to compute the similarity between two words A and B. Likewise, the similarity computations between APIs produced two $n \times n$ matrices, namely, malware and goodware API similarity matrix. Where n denotes the number of APIs in each category. Similarly, The similarity computations between system calls produced two $m \times m$ matrices, namely, malware and goodware system call similarity matrix. The index (i, j) in any matrix represents the semantic similarity score between two individual APIs or System calls i and j, respectively.

3.2. Feature Transformation phase

The idea behind feature transformation is to reduce the features' complexity and re-present them in a simplified form. According to our observation on the used features, the average length of an API or system call may surpass *twenty thousand* element, while the average number of permissions in applications is about *ten* per application. Therefore,

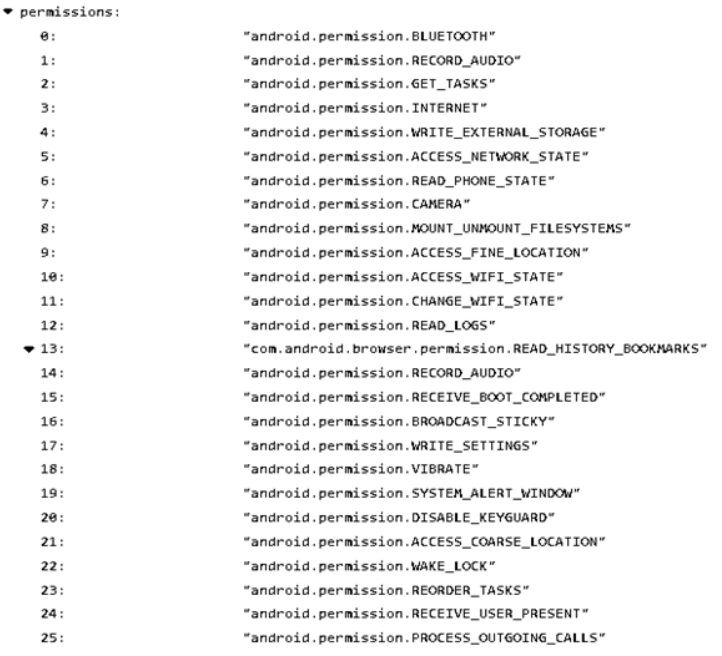

Fig. 2: Sample Extracted Permission List From Android Application

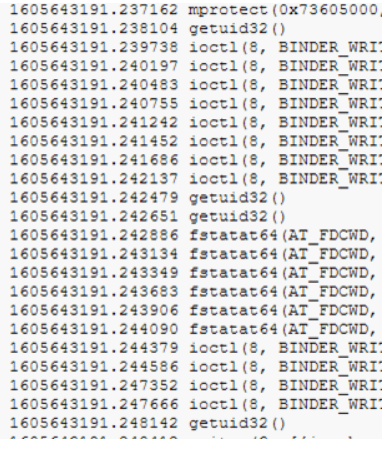

Fig. 3: Sample System call log For an Android Application

it might be beneficial to diminish the features' complexity in long sequences. Consequently, during this phase, we concentrated on transforming API and system calls.

The transformation process is performed by encapsulating contextually similar features into a limited number of clusters. Since the clusters bundle contextually relevant functions, they become more appropriate to express their contents. Consequently, we decided to transform the original calling sequences into clusters sequences. Hence, the cluster *id* is used to re-present the features that are indexed inside. Therefore, through a limited number of clusters, we can potentially limit the possible combinations of individual functions in malware or goodware calling sequences. As a result, our proposed model is not befuddled by a large number of features. In our experiments, our dataset contains more than 1 million distinct API functions and more than 250 distinct system calls. The API calls are encapsulated into *twenty* clusters, while the system calls are encapsulated in *seven* clusters.

To achieve the clustering step, the malware and goodware similarity matrices that were produced in section 3.1 are used as input to the k-means algorithm. Those matrices reflect the contextual or semantic similarity relations between

APIs and system functions in the whole malicious and non-malicious sequences. The clustering output, as shown in Fig. 1, are goodware and malware clusters that index APIs and system calls. We implemented a search function that searches for the API or system call in the clusters and returns the cluster *id* that indexes the API or system call. Therefore, the original calling sequence for API or system calls will be finally replaced by clusters *id*.

For example, the following snapshot is an API call subsequence for Android malware: *android.accounts.Account, android.accounts.Account.writeToParcel, android.accounts.AccountManager, android.accounts.AccountManager.get, android.animation.ObjectAnimator, android.animation.ObjectAnimator.addListener, android.animation.ObjectAnimator.cancel, android.animation.ObjectAnimator.ofFloat, android.animation.ObjectAnimator.setDuration, android.animation.ObjectAnimator.setInterpolator*.

Our search function will search each function inside the clusters and fetch the cluster *id* that contains it. According to our malware API indexing clusters, the above subsequence will be re-presented as: 1,15,2,15,17,12,17,12,17,12.

Moreover, the following snapshot is a system call subsequence of Android malware: *getuid, epoll_pwait, read, recvfrom, writev, sendto, writev, mprotect, ioctl, faccessat*. Similarly, according to our malware System call indexing clusters, the above subsequence will be re-presented as: 2,2,2,2,4,2,4,4,1,1. The output of feature transformation is a set of clusters sequences for malware and goodware API and system calls.

3.3. Classification phase

The main goal of the classification phase is to determine the behavior of a given process based on a given feature. The final classification decision decides whether the given process is malicious or not. Since the features as discussed in section 3.1 are different in length, We will consider two different classification methods that best suits the features' properties. Regarding the lengthy features such as API and system calls, we relied on the LSTM model to classify clusters' sequences that are generated from API and system calls. We chose the LSTM for training our model since it overcomes the RNN drawbacks such as the back-propagation gradient dispersion and gradient explosion. Hence, LSTM is more appropriate for the analysis of sequence data. Whilst concerning the permission feature, we experimented with different ML classifiers to form an ensemble voting model to classify permission lists.

The LSTM is a deep learning technique based on recurrent neural networks (RNNs). Since RNN was not effective enough in long-term learning, LSTM was produced. Any long-term dependence may be remembered and learned at random intervals using the LSTM architecture. Moreover, LSTM does not need any vectorization model, such as *tfidf* [55, 19]. LSTM added improvements over RNN through a specific module called constant error carousel (CEC), which propagates constant error signals over time, utilizing a well-designed "gate" structure to alleviate backpropagated errors from fading or growing. As it switches to manage the information flow and memory, the "gate" structure determines the internal value of CEC based on current input values and past context. A typical gate includes a pointwise multiplication operation and a nonlinear transformation, allowing mistakes to flow backward across a more extensive temporal range. The LSTM cell contains three gates: the input gate, the output gate, and the forget gate [58, 55].

Therefore, for any given an API or System call clusters sequence, $X = \{x_1, x_2, x_3, ..., x_t\}$, the input gate, forget gate, and output gate in the LSTM structure, are notated as i_t, f_t, and o_t, respectively. The weights and biases values associated to the above gates are W_i, W_f, W_o, b_i, b_f, b_o, respectively. Through every step, LSTM updates two states: the hidden state h_t, and the cell state c_t, and σ denotes the sigmoid function. The above parameters are presented as follows:

$$
\begin{aligned}
f_t &= \sigma\left(W_f * \left[h_{t-1}, x_t\right] + b_f\right) \\
i_t &= \sigma\left(W_i * \left[h_{t-1}, x_t\right] + b_i\right) \\
\widetilde{C}_t &= \tanh\left(W_c * \left[h_{t-1}, x_t\right] + b_c\right) \\
C_t &= f_t * C_{t-1} + i_t * \widetilde{C}_t \\
o_t &= \sigma\left(W_o * \left[h_{t-1}, x_t\right] + b_o\right), \\
h_t &= o_t * \tanh\left(C_t\right).
\end{aligned}
\tag{1}
$$

Yu et al. [59] showed that LSTM-layer stacking enhances the management of long-term dependencies in sequences. Correspondingly, LSTM is regarded as an effective technique for analysing data or occurrences with a precise connection, particularly in chronological sequence. In response, due to the inherently long-term dependencies

contained within the Android API and system calls, we choose to stack many LSTM layers together to capture the implicit relations between individual calls.

Our LSTM has three layers, the number of neurons in the first layer is set to 128, the second layer set to 64, and the third layer set to 32. We formulated the problem as a binary classification task. The network has been trained to predict a value of *zero* for regular sequences and a value of *one* for malicious sequences. To prevent overfitting, the training process is carried out through utilizing the ADAM optimizer with early stopping condition. We chose a learning rate of 0.001 and a batch size of 64.

The LSTM classification model shown in Fig. 1 receives malware and goodware clusters' sequences and associate a class label to the input sequence. However, We stated previously in section 2.4 that LSTM is not effective in learning from the extreme long sequence. Therefore, we overcome that situation by training our model over random short rapid-sequence snapshots of sizes range (100:1000) over the newly formed API and system calls sequences.

We used K-Fold cross-validation with (k=10) to train and test our LSTM classification models. Our training data is randomly partitioned into k-folds without replacement. During the training process, the LSTM models for API and system calls are trained using $k-1$ folds for the model training, and the remaining *one* fold is used for performance evaluation. We repeated the above procedure for k times; then, we got the average k performance estimates as our result.

Similarly, regarding permissions classification, we used 5-fold cross-validation to train and test the ML classifiers used to build an ensemble voting classifier. The training data set is divided into *five* parts where *four* parts are utilized for training the model, and *one* part is used to test the model. This technique is executed five times to ensure that each component is evaluated once. Subsequently, we get the average k performance estimates as our result. Finally, all the used classifiers are employed in a *hard* or majority voting mechanism to predict the target class.

4. Results and discussion

In this section, we assessed our model using a variety of datasets and conventional assessment measures. We demonstrated our model's capacity to recognize unknown malicious samples, especially the ransomware sequences.

4.1. Android Datasets

To evaluate the efficiency of our proposed approach in recognizing Android malware, we aggregated various heterogeneous datasets for training and testing. For example, we used static features datasets such as the permissions as well as dynamic features datasets like API, and system calls. In Table 3, we provided a summarized description for the employed Android datasets. Since our suggested model is intended to monitor and anticipate the behavioral execution of Android apps, we randomly picked *twenty* subsequences of a specific length from each API and system call sequence (if the entire sequence length permits).

Consequently, we split the API and system call training and testing datasets into different sized subsequences. Table 4 and Table 5 depict the samples for training and testing snapshots for API and system calls, respectively. Nowadays, with the growing number of ransomware attacks, it has become necessary for any security system to detect those types of new attacks. Therefore, we also provided sized snapshot samples in Table 6 for the goodware and ransomware datasets, which is considered a challenging evaluation for our model.

4.2. Evaluation metrics

We evaluated our model's performance using common evaluation measures such as precision, recall, accuracy, and F1-score (Equations 2-5). Additionally, we relied on false-positive rate (FPR) and false-negative rate (FNR) measures (Equations. 6-7). The latter measures are often used in the confusion matrix, which is a table that summarizes the classification algorithms' effectiveness.

$$Precision = \frac{TruePositives(TP)}{TP + FalsePositives(FP)} \tag{2}$$

$$Recall = \frac{TP}{TP + FalseNegatives(FN)} \tag{3}$$

Table 3
Android Malware and Goodware Training and Testing Datasets

Purpose	Dataset	Size		Description
		Malware	Goodware	
Training Datasets	Ban et al. (2018) [60]	30,920	30,810	API calls
	Martín et al. (2019) [3]	21,018	11,973	API calls
	Yerima et al. (2018) [61]	5,560	9,476	Permissions
	Akhilesh Sharma (2020) [62]	3.347	2.462	System Calls
Testing Datasets	Karbab et al. (2018) [27]	33,066	37,627	API calls
	Mahdavifar et al. (2020) [63]	1500	1500	Permissions
	Asma Razgallah (2021) [64]	200	200	System calls
	Chew et al. (2020) [65]	840	1506	System calls

Table 4
Android Malware and Goodware API Calls Training and Testing Snapshots samples

Call Size	Training		Testing	
	Malware	Goodware	Malware	Goodware
100	359,776	375,415	348,376	333,234
200	337,811	310,258	264,327	297,631
300	320,741	254,055	196,351	270,996
400	305,009	206,549	145,795	250,736
500	292,003	166,350	111,167	236,389
600	281,064	136,221	85,918	225,580
700	270,727	113,329	66,797	215,916
800	261,726	94,775	52,011	207,689
900	253,367	79,447	40,449	200,761
1000	246,791	67,076	31,281	194,351

Table 5
Android Malware and Goodware System Calls Training and Testing Snapshots samples

Call Size	Training		Testing	
	Malware	Goodware	Malware	Goodware
100	63,547	47,897	3,652	3,793
200	60,537	46,812	3,412	3,605
300	57,878	45,867	3,139	3,427
400	55,364	45,024	2,848	3,244
500	53,248	44,154	2,606	3,113
600	51,330	43,315	2,355	2,963
700	49,556	42,486	2,120	2,797
800	47,938	41,721	1,927	2,653
900	46,468	40,995	1,749	2,519
1000	45,044	40,174	1,597	2,376

$$F1 - Score = \frac{2 \times precision \times recall}{precision + recall} \tag{4}$$

$$Accuracy = \frac{TP + TrueNegatives(TN)}{TP + TN + FP + FN} \tag{5}$$

$$False\ Positive\ Rate = \frac{FP}{FP + TN} \tag{6}$$

Table 6
Testing Android ransomware System calls Snapshots Samples

Call Size	Training		Testing	
	Malware	Goodware	Malware	Goodware
100	63,547	47,897	12,619	30,050
200	60,537	46,812	12,539	29,972
300	57,878	45,867	12,477	29,899
400	55,364	45,024	12,398	29,835
500	53,248	44,154	12,324	29,757
600	51,330	43,315	12,264	29,682
700	49,556	42,486	12,199	29,625
800	47,938	41,721	12,135	29,567
900	46,468	40,995	12,082	29,494
1000	45,044	40,174	11,998	29,428

$$False\ Negative\ Rate = \frac{FN}{FN + TP} \tag{7}$$

4.3. Zero-day prediction evaluation based on API calls

According to the results that are shown in Table 7, we can observe an almost the same accuracy performance throughout each individual fold, despite the varying calling API sequence sizes. Note that because the used datasets are not balanced, we reported the mean accuracy and standard deviation over the 10-fold cross-validation as a representative metric of the model's performance. On average, we can observe a minor change in the average accuracy results with respect to different calling lengths. This observation does indeed provide proof that our model's accuracy was not affected by the length of the API call. The reason is that our model is not perplexed by the massive number of APIs that exist in the original calling sequence. Indeed, the replacement of individual APIs by the clusters that encapsulate them allowed our model to differentiate between malicious and non-malicious API sequences. Moreover, the sequence length does not affect the returned accuracy.

Our model also showed a distinguishable performance against the unseen API call sequence dataset. Since our sized-snaphots are not balanced, we also provided the standard deviation along with the average result per each accuracy measure. According to the results shown in Table 8, our model showed an average accuracy of 0.975 when monitoring random API call snapshots of size 100, with an average FPR and FNR values of 0.005 and 0.054, respectively. Our model accuracy was slightly enhanced when the snapshot call size increased. For example, at snapshots call size 200, results showed a slight proportional increase in terms of recall, accuracy, and FNR, with a minor enhancement in FNR value. We noticed that our system performance became almost stable when the snapshot calling length increased. According to Table 8, the average accuracy measures for calling size range (400:1000) are almost typical with some minor fractions.

On average, our model predicted the sequence behavior with an f-score and an accuracy value of 0.976. Moreover, it showed FPR and FNR values of 0.005 and 0.054, respectively. When comparing the average accuracy measures with the accuracy measures of different snapshot call sizes, we found that the snapshot API call size 400 almost returns similar accuracy results. Therefore, it is possible to use rapid API snapshots of size 400 to predict the whole sequence behavior.

4.4. Zero-day prediction evaluation based on system calls

According to results depicted in Table 9, our model showed considerable accuracy regarding the system call snapshot size in all folds. Similar to the situation in API calls, our new indexing mechanism reduced the association complexity among the original system calls. Therefore, instead of dealing with numerous associations of hundreds or thousands of features, we can express any sequence within a limited number of clusters that index those features. Consequently, we found an almost similar performance along the same fold despite the change in snapshot calling size.

We used two unseen datasets to evaluate our model using system calls as shown in Table 3. The first one is used in [64], where it contains 200 malware and 200 goodware system call sequences. As shown in Table 5, we generated random snapshots with several calling sizes for testing malware and goodware with no duplicate entries. Since our

Table 7
Testing accuracy of the 10-folds for API-calls snapshots of specified length

K-Fold/API-Calls	100-Call	200-Call	300-Call	400-Call	500-Call	600-Call	700-Call	800-Call	900-Call	1000-Call
1	0.993 ±0.003	0.991 ±0.002	0.990 ±0.004	0.992 ±0.002	0.991 ±0.003	0.991 ±0.003	0.990 ±0.002	0.992 ±0.003	0.990 ±0.003	0.990 ±0.002
2	0.993 ±0.003	0.993 ±0.003	0.992 ±0.002	0.993 ±0.003	0.993 ±0.003	0.994 ±0.002	0.993 ±0.004	0.990 ±0.003	0.992 ±0.003	0.993 ±0.003
3	0.994 ±0.004	0.993 ±0.002	0.993 ±0.003	0.991 ±0.004	0.993 ±0.003	0.995 ±0.003	0.994 ±0.002	0.992 ±0.003	0.993 ±0.002	0.994 ±0.003
4	0.993 ±0.002	0.993 ±0.002	0.992 ±0.002	0.994 ±0.002	0.993 ±0.004	0.993 ±0.002	0.992 ±0.003	0.991 ±0.003	0.992 ±0.002	0.992 ±0.004
5	0.994 ±0.002	0.994 ±0.003	0.992 ±0.004	0.992 ±0.004	0.991 ±0.004	0.993 ±0.004	0.993 ±0.003	0.989 ±0.004	0.992 ±0.005	0.993 ±0.003
6	0.995 ±0.002	0.993 ±0.002	0.994 ±0.003	0.992 ±0.003	0.992 ±0.002	0.994 ±0.004	0.994 ±0.003	0.991 ±0.003	0.994 ±0.004	0.994 ±0.003
7	0.995 ±0.002	0.995 ±0.002	0.994 ±0.003	0.992 ±0.004	0.992 ±0.003	0.995 ±0.004	0.994 ±0.002	0.991 ±0.003	0.994 ±0.003	0.994 ±0.002
8	0.991 ±0.002	0.991 ±0.002	0.994 ±0.002	0.994 ±0.004	0.994 ±0.004	0.993 ±0.004	0.993 ±0.004	0.993 ±0.003	0.994 ±0.004	0.993 ±0.004
9	0.995 ±0.002	0.992 ±0.002	0.993 ±0.002	0.993 ±0.003	0.992 ±0.005	0.994 ±0.005	0.994 ±0.004	0.993 ±0.003	0.993 ±0.004	0.994 ±0.004
10	0.994 ±0.002	0.995 ±0.002	0.993 ±0.002	0.994 ±0.004	0.994 ±0.005	0.993 ±0.005	0.994 ±0.004	0.994 ±0.004	0.993 ±0.003	0.994 ±0.003
Average	**0.994**	**0.993**	**0.993**	**0.993**	**0.993**	**0.994**	**0.993**	**0.992**	**0.993**	**0.993**

Table 8
Prediction accuracy for unseen API call testing Dataset based on API call snapshot length

Call Size	Accuracy Measures					
	Precision	Recall	F1-Score	Accuracy	FPR	FNR
100	0.992 ±0.003	0.946 ±0.014	0.968 ±0.010	0.975 ±0.010	0.005 ±0.002	0.054 ±0.014
200	0.992 ±0.002	0.947 ±0.014	0.969 ±0.008	0.976 ±0.008	0.005 ±0.002	0.053 ±0.014
300	0.993 ±0.002	0.943 ±0.013	0.967 ±0.008	0.974 ±0.008	0.004 ±0.001	0.057 ±0.013
400	0.993 ±0.002	0.947 ±0.013	0.969 ±0.007	0.976 ±0.007	0.005 ±0.002	0.053 ±0.013
500	0.991 ±0.002	0.948 ±0.012	0.970 ±0.007	0.976 ±0.007	0.005 ±0.001	0.052 ±0.012
600	0.992 ±0.002	0.944 ±0.011	0.968 ±0.006	0.975 ±0.006	0.005 ±0.001	0.056 ±0.011
700	0.991 ±0.002	0.947 ±0.010	0.969 ±0.006	0.976 ±0.006	0.006 ±0.001	0.053 ±0.010
800	0.993 ±0.002	0.948 ±0.010	0.970 ±0.006	0.976 ±0.006	0.005 ±0.002	0.052 ±0.010
900	0.992 ±0.002	0.945 ±0.009	0.968 ±0.005	0.975 ±0.005	0.005 ±0.001	0.055 ±0.009
1000	0.992 ±0.002	0.944 ±0.009	0.968 ±0.005	0.974 ±0.005	0.005 ±0.001	0.056 ±0.009
Average	0.992	0.945	0.968	0.976	0.005	0.054

System calls sized-snaphots are not balanced, we provided the standard deviation along with the average result per each accuracy measure.

As demonstrated by Table 10, the average precision, recall values are enhanced with a ratio of almost ($\approx 1\% - 1.5\%$) during the progressive increase of calling size. Correspondingly, average the f1-score and accuracy values are increased by that ratio too. We noticed that the average accuracy values for system call snapshots (400–1000) are almost similar with minor fractions. Accordingly, with a system call snapshot of only size 400, our system can distinguish the unknown system calling sequence with an accuracy of 97%.

The second unseen system call dataset used in [65] contains a system call logs for ransomware *WannaLocker, SimpleLocker, WipeLocker, Pletor, FileCoder, and Black Rose Lucy*. According to the results described in Table 11 with only snapshot-size 100, our system can predict with an average accuracy of 0.976 that the system calling sequence is ransomware or not. The accuracy results are slightly enhanced while the snapshot calling size increases. Nevertheless, the overall accuracy enhancement ratio wasn't much increased. However, according to the average accuracy values for different accuracy measures, we can conclude that, with a snapshot calling size 400, we found that our model performs almost like the average accuracy results. Consequently, likewise our previous experiments, we can conclude that a rapid system call snapshots of length 400 is enough to determine the sequence behavior. In Fig. 4, we provided a summarized description regarding the prediction accuracy of our model in identifying unseen system calls and ransomware samples.

4.5. Zero-Day Prediction Evaluation Based on Permissions

As discussed in section 3.3, we formed our permission ensemble classifier through training a common collection of ML classifiers. Our chosen ensemble algorithm is the simple majority voting algorithm. Therefore, an odd number of voter classifiers was required. The best-performing classifiers were then chosen to create our ensemble classification model. The classifier selection criteria is based on each classifier's unique accuracy performance. As a result, we combined our permission training dataset [61] with several ML classification algorithms, including Decision Tree, Random Forest, Extra Tree, Support Vector Machine (SVM), k-nearest neighbor (KNN), Linear Discriminant Analysis (LDA), Multilayer Perceptron (MLP), Adaptive Boosting (AdaBoost), and Naive Bayes (NB). In our experiment, we used the *median* value as a threshold selection criteria for the candidate classifiers.

Table 9
Testing accuracy for k-fold accuracy for system-calls snapshots of specified length

K-Fold/System-Calls	100-Call	200-Call	300-Call	400-Call	500-Call	600-Call	700-Call	800-Call	900-Call	1000-Call
1	0.996 ±0.003	0.998 ±0.003	0.996 ±0.002	1.000 ±0.002	0.994 ±0.004	0.999 ±0.004	0.996 ±0.003	0.998 ±0.003	0.998 ±0.002	0.997 ±0.003
2	0.995 ±0.003	0.998 ±0.004	0.996 ±0.004	1.000 ±0.003	0.994 ±0.004	0.999 ±0.005	0.996 ±0.003	0.998 ±0.004	0.998 ±0.004	0.996 ±0.003
3	0.996 ±0.004	0.998 ±0.005	0.996 ±0.005	1.000 ±0.004	0.994 ±0.004	0.999 ±0.005	0.996 ±0.005	0.998 ±0.004	0.998 ±0.005	0.997 ±0.004
4	0.996 ±0.003	0.999 ±0.004	0.996 ±0.004	1.000 ±0.004	0.994 ±0.004	0.999 ±0.004	0.996 ±0.003	0.998 ±0.004	0.998 ±0.004	0.997 ±0.005
5	0.996 ±0.003	0.998 ±0.003	0.996 ±0.004	1.000 ±0.004	0.994 ±0.005	0.999 ±0.003	0.996 ±0.004	0.998 ±0.003	0.998 ±0.004	0.997 ±0.003
6	0.996 ±0.005	0.998 ±0.004	0.996 ±0.005	1.000 ±0.004	0.994 ±0.006	0.999 ±0.005	0.996 ±0.005	0.998 ±0.006	0.998 ±0.005	0.997 ±0.005
7	0.996 ±0.004	0.998 ±0.003	0.996 ±0.005	1.000 ±0.005	0.994 ±0.004	0.999 ±0.005	0.997 ±0.006	0.998 ±0.005	0.998 ±0.004	0.997 ±0.005
8	0.995 ±0.005	0.998 ±0.004	0.996 ±0.006	1.000 ±0.005	0.994 ±0.004	0.999 ±0.005	0.996 ±0.004	0.998 ±0.005	0.998 ±0.005	0.997 ±0.006
9	0.996 ±0.004	0.998 ±0.003	0.996 ±0.005	1.000 ±0.004	0.994 ±0.006	0.999 ±0.004	0.996 ±0.006	0.998 ±0.004	0.998 ±0.005	0.997 ±0.005
10	0.996 ±0.005	0.998 ±0.003	0.996 ±0.006	1.000 ±0.004	0.994 ±0.005	0.999 ±0.004	0.996 ±0.005	0.998 ±0.004	0.998 ±0.005	0.996 ±0.005
Average	**0.996**	**0.998**	**0.996**	**1.000**	**0.994**	**0.999**	**0.996**	**0.998**	**0.998**	**0.997**

Table 10
Prediction accuracy for unseen System calls testing Dataset based on System call snapshot length

Call Size	Accuracy Measures					
	Precision	Recall	F1-Score	Accuracy	FPR	FNR
100	0.982 ±0.004	0.932 ±0.003	0.957 ±0.003	0.956 ±0.003	0.018 ±0.003	0.068 ±0.003
200	0.987 ±0.003	0.935 ±0.002	0.960 ±0.002	0.960 ±0.002	0.013 ±0.003	0.065 ±0.002
300	0.992 ±0.004	0.941 ±0.004	0.966 ±0.004	0.967 ±0.005	0.007 ±0.004	0.059 ±0.004
400	0.996 ±0.005	0.944 ±0.005	0.969 ±0.005	0.970 ±0.006	0.004 ±0.005	0.056 ±0.005
500	0.997 ±0.006	0.943 ±0.005	0.969 ±0.005	0.971 ±0.006	0.002 ±0.006	0.057 ±0.005
600	0.999 ±0.006	0.942 ±0.004	0.970 ±0.005	0.972 ±0.006	0.001 ±0.006	0.058 ±0.004
700	1.000 ±0.006	0.942 ±0.004	0.970 ±0.005	0.973 ±0.006	0.000 ±0.006	0.058 ±0.004
800	0.999 ±0.006	0.939 ±0.004	0.968 ±0.005	0.972 ±0.006	0.000 ±0.006	0.061 ±0.004
900	0.999 ±0.006	0.938 ±0.004	0.968 ±0.004	0.973 ±0.006	0.000 ±0.006	0.062 ±0.004
1000	0.999 ±0.006	0.939 ±0.004	0.968 ±0.004	0.974 ±0.006	0.000 ±0.006	0.061 ±0.004
Average	0.995	0.940	0.967	0.969	0.005	0.060

Table 11
Ransomware malware prediction accuracy based on system call snapshot length

Call Size	Accuracy Measures					
	Precision	Recall	F1-Score	Accuracy	FPR	FNR
100	0.952 ±0.004	0.979 ±0.002	0.965 ±0.003	0.980 ±0.003	0.020 ±0.002	0.021 ±0.002
200	0.959 ±0.004	0.982 ±0.002	0.970 ±0.003	0.983 ±0.002	0.017 ±0.002	0.018 ±0.002
300	0.962 ±0.004	0.984 ±0.002	0.973 ±0.003	0.984 ±0.002	0.016 ±0.002	0.016 ±0.002
400	0.967 ±0.005	0.984 ±0.002	0.975 ±0.004	0.986 ±0.002	0.014 ±0.002	0.016 ±0.002
500	0.967 ±0.006	0.985 ±0.002	0.976 ±0.004	0.986 ±0.002	0.013 ±0.002	0.015 ±0.002
600	0.970 ±0.005	0.985 ±0.002	0.977 ±0.004	0.987 ±0.002	0.012 ±0.003	0.015 ±0.002
700	0.970 ±0.005	0.986 ±0.002	0.978 ±0.004	0.987 ±0.002	0.012 ±0.003	0.014 ±0.002
800	0.971 ±0.005	0.987 ±0.002	0.979 ±0.004	0.988 ±0.002	0.012 ±0.003	0.013 ±0.002
900	0.971 ±0.005	0.988 ±0.003	0.980 ±0.005	0.988 ±0.002	0.012 ±0.003	0.012 ±0.002
1000	0.972 ±0.005	0.989 ±0.003	0.980 ±0.005	0.989 ±0.003	0.011 ±0.003	0.011 ±0.002
Average	0.966	0.985	0.975	0.986	0.014	0.015

We utilized the k-fold cross-validation with k = 5 to train each classifier in our classifiers. Our training data is randomly partitioned into 5-folds without replacement. Our permission training dataset is split into five parts (four parts for training and one part for testing). This previous technique is performed five times to ensure that each component is evaluated once. Table 12 outlines the individual accuracy for each classifier.

According to the results described in Table 12, based on the calculated median value, we selected all classifiers that have accuracy value greater than or equal to the median value. According to our selection criteria, we chose five classifiers to establish our ensemble voting model. Therefore, our permission ensemble classifier model is constructed using the Random forest, MLP, AdaBoost, SVM, and Decision Tree classifiers. Eventually, all selected classifiers are employed in a hard voting mechanism to predict the target class.

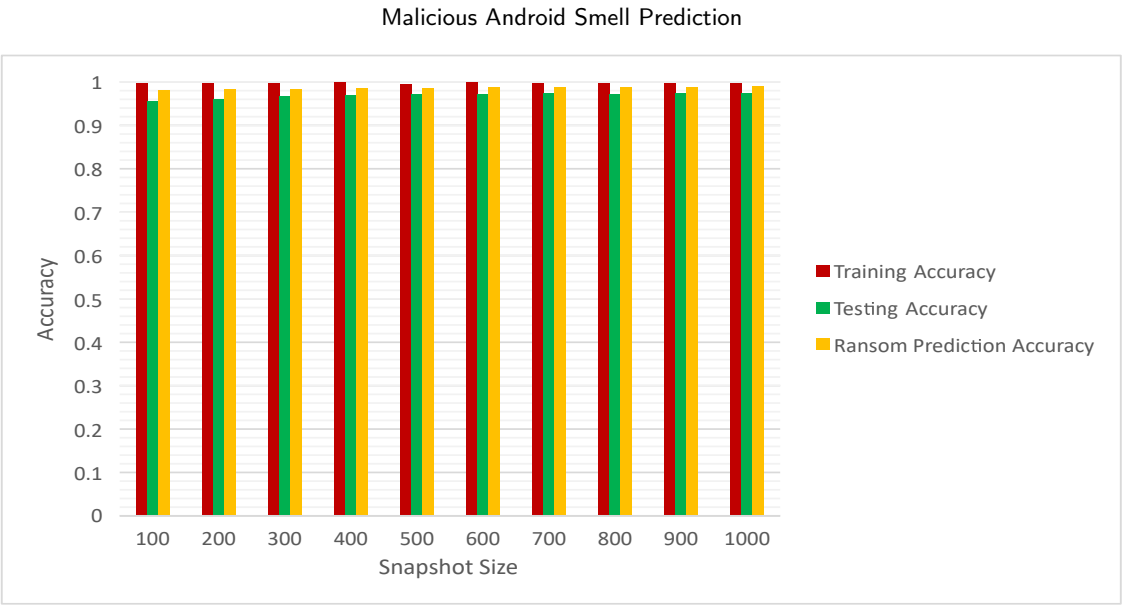

Fig. 4: System Calls Training Accuracy vs Unseen Testing and Ransomware Prediction Accuracy

Table 12
Candidate classifiers average accuracy

Rank	Classifier	Accuracy
1	Random Forest	0.989
2	MLP	0.988
3	AdaBoost	0.984
4	SVM	0.983
5	Decision Tree	0.976
6	Extra Tree	0.975
7	LDA	0.918
8	KNN	0.916
9	NB	0.761
Median		**0.976**

We evaluated our ensemble model using the unseen permission dataset used in [63] that was previously described in Table 3. We extracted Apps permissions from 1500 malicious applications and 1500 goodware ones. Moreover, we evaluated our ensemble voting model against each classifier in Table 12. In terms of accuracy, Table 13 denoted that our formed ensemble classifier showed f-score and accuracy values of 0.993, which outperformed all other individual classifiers. Our ensemble model also showed the lowest FPR and FNR, with values of 0.007. Those accuracy measures reveal the efficiency of our ensemble in distinguishing permissions' behavior in different applications.

In comparison with other Android malware detection approaches, our proposed model showed a competitive performance compared to other peer approaches as shown in Table 14. Our model showed a prediction F-score and accuracy values of 0.990 and 0.993, respectively, relying on permissions. Moreover, it showed a distinguishable prediction accuracy of 0.976 when relying on API calls. Furthermore, it showed a prediction accuracy of 0.969 when our model employed system calls. The accuracy results depicted in Table 14 validated our proposed model's capacity to interpret the behavior of an Android process. Accordingly, we can early detect and stop malicious processes in advance.

5. Challenges and future directions

According to our experimental work, we can assure that the conventional techniques used for malware detection become less effective. Therefore, we have to cope with the revolution of the malware industry with more intelligent

Table 13
Individual Classifier Detection Accuracy for Android Permissions over Unseen Testing Dataset

Classifier	Accuracy Measures					
	Precision	Recall	F1-Score	Accuracy	FPR	FNR
Rabdom Forest	0.989	0.954	0.971	0.971	0.012	0.013
MLP	0.966	0.983	0.975	0.975	0.033	0.017
AdaBoost	0.987	0.981	0.984	0.984	0.013	0.013
SVM	0.965	0.994	0.979	0.980	0.034	0.006
Decision Tree	0.974	0.991	0.982	0.983	0.026	0.026
Extra Tree	0.933	0.974	0.953	0.954	0.064	0.026
LDA	0.881	0.953	0.915	0.919	0.119	0.111
KNN	0.817	0.994	0.897	0.906	0.183	0.156
NB	0.502	0.642	0.563	0.611	0.409	0.280
Our Ensemble	0.993	0.992	0.993	0.993	0.007	0.007

Table 14
Comparison with other research works regarding android malware detection

Study	Number of Malware	F1-Score	Accuracy	Feature used
Zhenlong et al. (2016) [66]	1,760	0.968	-	Multi features
Xu et al. (2016) [67]	5,888	-	0.934	Multi features
Hou et al. (2017) [26]	2,500	0.966	0.966	API calls
Liang et al. (2017) [68]	14,231	0.866	0.931	system calls
Wang et al. (2018) [69]	6,334	-	0.945	API calls and permissions
Kim et al. (2018) [70]	13,075	-	0.980	Multi features
Zhu et al. (2018) [21]	2,130	-	0.883	API calls and permissions
Zhu et al. (2018) [71]	10,770	0.974	0.974	Multi features
Karbab et al. (2018)) [27]	33,066	-	0.962	API calls
Hasegawa et al. (2018) [72]	7,000	-	0.954	end-to-end
Ren et al. (2020) [73]	16,000	0.958	0.958	end-to-end
Alzaylaee et al.(2020) [7]	11,505	0.962	0.952	Permissions
D'Angelo et al.(2020) [28]	1750	0.970	0.940	API calls
Zhang et al.(2021) [74]	5000	0.966	0.966	Multi features
Gao et al.(2021) [75]	1200	0.960	0.969	API calls
Sasidharan et al.(2021) [76]	1500	-	0.945	API calls
Amer et al.(2021) [17]	33,066	0.977	0.976	API calls
Proposed Model	1500	0.990	0.993	Permission
	33,066	0.968	0.976	API call
	1040	0.967	0.969	System Call

and robust solutions. Due to current malware's abundance, aggression, and complication, especially zero-day attacks, malware analysis techniques should now transit from malware detection into malware prediction. The IoT's rapid expansion and numerous uses raise the danger of malware threats. Due to the several capabilities of IoT devices and the dynamic and ever-changing environment, adhering to basic security methods becomes dangerous, and implementing comprehensive security measures becomes mandatory.

In future work, we aimed at generating a signature benchmark feature for malicious and non-malicious processes. The benchmark future will be generated through the fusing of heterogeneous significant features into a single representative feature. Therefore, through smart analysis to the fused feature, we can discover the discriminators between malicious and non-malicious processes.

6. Conclusion

Cybersecurity has developed into a serious worry for society, given the rising frequency of cyber-attacks and the breadth of their targets. Android devices are considered an entry point for various attack strategies due to their prevalence. Therefore, Android users are continuously at risk of being harmed by an ever-growing number of malicious

apps. As a result, predicting malware variants is a significant issue, given the potential for damage and the rapid rate at which new malware variants emerge. In this paper, we introduced a model that analyses malicious Android processes. We used hybrid analysis features to analyze Android apps in our model. We demonstrated that by analyzing random short snapshots of processes, we could forecast their harmful behavior. We also proved with a comparative accuracy measures the proficiency of our model against unseen samples, including ransomware samples. In conclusion, our model was not trapped in the massive and complex feature association; instead, we dealt with feature complications efficiently, allowing us to capture many malicious patterns. In future work, we aim to incorporate other behavioral-driven heuristics to keep our models adaptive against new malware threats.

References

[1] L. Nguyen-Vu, J. Ahn, and S. Jung, "Android fragmentation in malware detection," *Computers & Security*, vol. 87, p. 101573, 2019.

[2] A. Qamar, A. Karim, and V. Chang, "Mobile malware attacks: Review, taxonomy & future directions," *Future Generation Computer Systems*, vol. 97, pp. 887–909, 2019.

[3] A. Martín, R. Lara-Cabrera, and D. Camacho, "Android malware detection through hybrid features fusion and ensemble classifiers: the andropytool framework and the omnidroid dataset," *Information Fusion*, vol. 52, pp. 128–142, 2019.

[4] M. Wazid, S. Zeadally, and A. K. Das, "Mobile banking: evolution and threats: malware threats and security solutions," *IEEE Consumer Electronics Magazine*, vol. 8, no. 2, pp. 56–60, 2019.

[5] E. Alepis and C. Patsakis, "Hey doc, is this normal?: exploring android permissions in the post marshmallow era," in *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 2017, pp. 53–73.

[6] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. Gaur, M. Conti, and R. Muttukrishnan, "Android security: A survey of issues," *Malware Penetration and Defenses*.

[7] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Dl-droid: Deep learning based android malware detection using real devices," *Computers & Security*, vol. 89, p. 101663, 2020.

[8] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: a survey of issues, malware penetration, and defenses," *IEEE communications surveys & tutorials*, vol. 17, no. 2, pp. 998–1022, 2014.

[9] S. Wang, Z. Chen, Q. Yan, K. Ji, L. Peng, B. Yang, and M. Conti, "Deep and broad url feature mining for android malware detection," *Information Sciences*, vol. 513, pp. 600–613, 2020.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[11] A. De Lorenzo, F. Martinelli, E. Medvet, F. Mercaldo, and A. Santone, "Visualizing the outcome of dynamic analysis of android malware with vizmal," *Journal of Information Security and Applications*, vol. 50, p. 102423, 2020.

[12] I. Zelinka and E. Amer, "An ensemble-based malware detection model using minimum feature set," in *MENDEL*, vol. 25, no. 2, 2019, pp. 1–10.

[13] E. Amer and I. Zelinka, "A dynamic windows malware detection and prediction method based on contextual understanding of api call sequence," *Computers & Security*, vol. 92, p. 101760, 2020.

[14] H. Cai, X. Fu, and A. Hamou-Lhadj, "A study of run-time behavioral evolution of benign versus malicious apps in android," *Information and Software Technology*, vol. 122, p. 106291, 2020.

[15] E. Amer, S. El-Sappagh, and J. W. Hu, "Contextual identification of windows malware through semantic interpretation of api call sequence," *Applied Sciences*, vol. 10, no. 21, p. 7673, 2020.

[16] V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. de Geus, "Identifying android malware using dynamically obtained features," *Journal of Computer Virology and Hacking Techniques*, vol. 11, no. 1, pp. 9–17, 2015.

[17] E. Amer, I. Zelinka, and S. El-Sappagh, "A multi-perspective malware detection approach through behavioral fusion of api call sequence," *Computers & Security*, vol. 110, p. 102449, 2021.

[18] A. Mathur, L. M. Podila, K. Kulkarni, Q. Niyaz, and A. Y. Javaid, "Naticusdroid: A malware detection framework for android using native and custom permissions," *Journal of Information Security and Applications*, vol. 58, p. 102696, 2021.

[19] A. Razgallah, R. Khoury, S. Hallé, and K. Khanmohammadi, "A survey of malware detection in android apps: Recommendations and perspectives for future research," *Computer Science Review*, vol. 39, p. 100358, 2021.

[20] A. Arora, S. K. Peddoju, and M. Conti, "Permpair: Android malware detection using permission pairs," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1968–1982, 2019.

[21] H.-J. Zhu, Z.-H. You, Z.-X. Zhu, W.-L. Shi, X. Chen, and L. Cheng, "Droiddet: effective and robust detection of android malware using static analysis along with rotation forest model," *Neurocomputing*, vol. 272, pp. 638–646, 2018.

[22] E. Amer, "Permission-based approach for android malware analysis through ensemble-based voting model," in *2021 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*. IEEE, 2021, pp. 135–139.

[23] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell, "Androdialysis: Analysis of android intent effectiveness in malware detection," *computers & security*, vol. 65, pp. 121–134, 2017.

[24] D. Ucci, L. Aniello, and R. Baldoni, "Survey of machine learning techniques for malware analysis," *Computers & Security*, vol. 81, pp. 123–147, 2019.

[25] A. Pektaş and T. Acarman, "Deep learning for effective android malware detection using api call graph embeddings," *Soft Computing*, vol. 24, no. 2, pp. 1027–1043, 2020.

[26] 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 2017*, 2017, pp. 803–810.

[27] 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.

[28] G. D'Angelo, M. Ficco, and F. Palmieri, "Malware detection in mobile environments based on autoencoders and api-images," *Journal of Parallel and Distributed Computing*, vol. 137, pp. 26–33, 2020.

[29] H. Zhang, S. Luo, Y. Zhang, and L. Pan, "An efficient android malware detection system based on method-level behavioral semantic analysis," *IEEE Access*, vol. 7, pp. 69 246–69 256, 2019.

[30] M. Qiao, A. H. Sung, and Q. Liu, "Merging permission and api features for android malware detection," in *2016 5th IIAI international congress on advanced applied informatics (IIAI-AAI)*. IEEE, 2016, pp. 566–571.

[31] 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*. Springer, 2014, pp. 163–182.

[32] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version)," *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 2, pp. 1–34, 2019.

[33] R. Taheri, M. Ghahramani, R. Javidan, M. Shojafar, Z. Pooranian, and M. Conti, "Similarity-based android malware detection using hamming distance of static binary features," *Future Generation Computer Systems*, vol. 105, pp. 230–247, 2020.

[34] S. Gunasekera and M. Thomas, *Android apps security*. Springer, 2012.

[35] F. Ou and J. Xu, "S3feature: A static sensitive subgraph-based feature for android malware detection," *Computers & Security*, p. 102513, 2021.

[36] T. Bhatia and R. Kaushal, "Malware detection in android based on dynamic analysis," in *2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*. IEEE, 2017, pp. 1–6.

[37] W. Yu, L. Ge, G. Xu, and X. Fu, "Towards neural network based malware detection on android mobile devices," in *Cybersecurity Systems for Human Cognition Augmentation*. Springer, 2014, pp. 99–117.

[38] X. Xiao, Z. Wang, Q. Li, S. Xia, and Y. Jiang, "Back-propagation neural network on markov chains from system call sequences: a new approach for detecting android malware with system call sequences," *IET Information Security*, vol. 11, no. 1, pp. 8–15, 2017.

[39] M. Dimjašević, S. Atzeni, I. Ugrina, and Z. Rakamaric, "Evaluation of android malware detection based on system calls," in *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, 2016, pp. 1–8.

[40] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Detecting android malware using sequences of system calls," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, 2015, pp. 13–20.

[41] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 83–97, 2016.

[42] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah, "Android malware detection based on system call sequences and lstm," *Multimedia Tools and Applications*, vol. 78, no. 4, pp. 3979–3999, 2019.

[43] M. Alazab and M. Tang, *Deep learning applications for cyber security*. Springer, 2019.

[44] J. Zhu, Z. Wu, Z. Guan, and Z. Chen, "Api sequences based malware detection for android," in *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. IEEE, 2015, pp. 673–676.

[45] M. Sun, X. Li, J. C. Lui, R. T. Ma, and Z. Liang, "Monet: a user-oriented behavior-based malware variants detection system for android," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 5, pp. 1103–1112, 2016.

[46] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *International conference on security and privacy in communication systems*. Springer, 2013, pp. 86–103.

[47] A. Skovoroda and D. Gamayunov, "Automated static analysis and classification of android malware using permission and api calls models," in *2017 15th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2017, pp. 243–24 309.

[48] L. Onwuzurike, M. Almeida, E. Mariconti, J. Blackburn, G. Stringhini, and E. De Cristofaro, "A family of droids-android malware detection via behavioral modeling: Static vs dynamic analysis," in *2018 16th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2018, pp. 1–10.

[49] S. Millar, N. McLaughlin, J. M. del Rincon, and P. Miller, "Multi-view deep learning for zero-day android malware detection," *Journal of Information Security and Applications*, vol. 58, p. 102718, 2021.

[50] R. Kumar, Z. Xiaosong, R. U. Khan, J. Kumar, and I. Ahad, "Effective and explainable detection of android malware based on machine learning algorithms," in *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence*, 2018, pp. 35–40.

[51] K. Khanmohammadi, N. Ebrahimi, A. Hamou-Lhadj, and R. Khoury, "Empirical study of android repackaged applications," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3587–3629, 2019.

[52] J. Jung, H. Kim, D. Shin, M. Lee, H. Lee, S.-j. Cho, and K. Suh, "Android malware detection based on useful api calls and machine learning," in *2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*. IEEE, 2018, pp. 175–178.

[53] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer, "Emulator vs real phone: Android malware detection using machine learning," in *Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics*, 2017, pp. 65–72.

[54] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, "Malware classification with recurrent networks," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 1916–1920.

[55] J. Yan, Y. Qi, and Q. Rao, "Lstm-based hierarchical denoising network for android malware detection," *Security and Communication Networks*, vol. 2018, 2018.

[56] A. Sheneamer, S. Roy, and J. Kalita, "A detection framework for semantic code clones and obfuscated code," *Expert Systems with Applications*, vol. 97, pp. 405–420, 2018.

[57] X. Ma, S. Guo, W. Bai, J. Chen, S. Xia, and Z. Pan, "An api semantics-aware malware detection method based on deep learning," *Security and Communication Networks*, vol. 2019, 2019.

[58] A. Sherstinsky, "Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network," *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, 2020.

[59] Y. Yu, X. Si, C. Hu, and J. Zhang, "A review of recurrent neural networks: Lstm cells and network architectures," *Neural computation*, vol. 31, no. 7, pp. 1235–1270, 2019.

[60] T. Ban, T. Takahashi, S. Guo, D. Inoue, and K. Nakao, "Integration of multi-modal features for android malware detection using linear svm," in *2016 11th Asia Joint Conference on Information Security (AsiaJCIS)*. IEEE, 2016, pp. 141–146.

[61] S. Y. Yerima and S. Sezer, "Droidfusion: A novel multilevel classifier fusion approach for android malware detection," *IEEE transactions on cybernetics*, vol. 49, no. 2, pp. 453–466, 2018.

[62] A. Sharm, "Android System Calls Dataset," https://github.com/Akhilesh64/Android-Malware-Detection, 2020, accessed: 2021-07-30.

[63] S. Mahdavifar, A. F. A. Kadir, R. Fatemi, D. Alhadidi, and A. A. Ghorbani, "Dynamic android malware category classification using semi-supervised deep learning," in *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE, 2020, pp. 515–522.

[64] A. Razgallah, "TwinDroid-dataset," https://github.com/AsmaLif/TwinDroid-dataset, 2021, accessed: 2021-07-30.

[65] C. J.-W. Chew, V. Kumar, P. Patros, and R. Malik, "Escapade: Encryption-type-ransomware: System call based pattern detection," in *International Conference on Network and System Security*. Springer, 2020, pp. 388–407.

[66] 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.

[67] L. Xu, D. Zhang, N. Jayasena, and J. Cavazos, "Hadm: Hybrid analysis for detection of malware," in *Proceedings of SAI Intelligent Systems Conference*. Springer, 2016, pp. 702–724.

[68] H. Liang, Y. Song, and D. Xiao, "An end-to-end model for android malware detection," in *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*. IEEE, 2017, pp. 140–142.

[69] Z. Wang, J. Cai, S. Cheng, and W. Li, "Droiddeeplearner: Identifying android malware using deep learning," in *2016 IEEE 37th Sarnoff Symposium*. IEEE, 2016, pp. 160–165.

[70] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "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.

[71] Y. Zhang, Y. Yang, and X. Wang, "A novel android malware detection approach based on convolutional neural network," in *Proceedings of the 2nd International Conference on Cryptography, Security and Privacy*, 2018, pp. 144–149.

[72] C. Hasegawa and H. Iyatomi, "One-dimensional convolutional neural networks for android malware detection," in *2018 IEEE 14th International Colloquium on Signal Processing & Its Applications (CSPA)*. IEEE, 2018, pp. 99–102.

[73] 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, p. 102098, 2020.

[74] N. Zhang, Y.-a. Tan, C. Yang, and Y. Li, "Deep learning feature exploration for android malware detection," *Applied Soft Computing*, vol. 102, p. 107069, 2021.

[75] H. Gao, S. Cheng, and W. Zhang, "Gdroid: Android malware detection and classification with graph convolutional network," *Computers & Security*, vol. 106, p. 102264, 2021.

[76] S. K. Sasidharan and C. Thomas, "Prodroid—an android malware detection framework based on profile hidden markov model," *Pervasive and Mobile Computing*, vol. 72, p. 101336, 2021.