Hybrid detection model with exploration approach for Android malware detection

Damien Strullu
60500 Chantilly, France
damien.strullu@gmail.com

Abstract

With the growth of malware usage on mobile platforms, practitioners and researchers have attempted to provide effective methods for producing suitable malware detection models. Over the last few decades, several such models have been based on program analysis (static and/or dynamic). However, modern hackers have created malware that is able to avoid detection by applying evasion techniques. In view of this, we introduce a new approach to dealing with these evasion techniques. To achieve this, we perform both static and dynamic analyses, and between these two steps, our model applies filtering based on call graph mining, which is executed during the static analysis. Our model provides detection with a respectable degree of accuracy of close to 0.98 and an F1-score of 0.95, while avoiding any waste of time arising from a dynamic analysis with limited scope, and offers Boolean detection.

Keywords

reverse engineering, malware detection, dynamic analysis, static analysis, hybrid analysis, android, mobile malware, machine learning.

Declaration

The dataset produced in this work is available on this repository: https://github.com/Dam4323/EGDataset. The model use in this study isn't directly available but you can contact the author to submit a request (preferably by e-mail) if you want to use it in your work. This research was fully funded by personal funds.

Introduction

Every day, news broadcasts report information about data leakages, criminal activities, or global breakdowns in which malware is implicated. In the last month, there have been many examples of banking fraud using trojans or ransomware. In the recent past, these criminal activities did not use informatic programs, as critical services such as banking services were not easily accessible to customers. Today, these services are always connected to customers via numerous devices (computers, smartphones, the IoT etc.). Smartphones represent a critical target, as this type of device is always activated and connected to the Internet to ensure its functioning (sending messages, receiving data from SDNs etc.). As we can see, this is a topic of intense research interest.

In this paper, we focus our attention on malware analysis techniques. Over the last two decades, many researchers have been working in the area of model analysis. All of these models are based on the two principal techniques of static and dynamic analysis, and we will introduce these concepts later. Research on this topic has evolved very quickly, and many tools have been developed to perform malware detection. These tools use a wide range of mathematical theories, such as graph theory, machine learning, classification, data modelling, deep learning and others. To differentiate between these models, we consider two dimensions: the metrics used by the model, and the analysis approach that is implemented.

Features: These are used to measure certain aspects of malware behaviour, and are numerous. We can classify these into two types: syntactic and resource-centric [7]. In general, a model extracts certain features to construct metrics, and focuses on a single type of metric. The earliest models used API calls to evaluate potential suspect behaviour [1,2,4]. These models were based on graph analysis and semantic analysis, but they had some limitations, especially when programs were obfuscated. Some improvements have been made to avoid this limitation, notably semi-level language design known as aposcopy [15]. Each year, researchers introduce further

types of metrics, such as app structure [1], network data flow [1], CPU consumption [5], function calls [1,4] and system events. The choice made by a researcher determines the process and the approach used in a model.

Approach: This represents the logical path implemented in the model. The initial models used a single analysis technique (dynamic or static), but over the last decade, researchers have combined these two techniques to avoid some of the limitations encountered by the old models. Their limitations include the following: data extraction is impossible due to the obfuscation of binaries [1,17]; there is a high proportion of true positives, especially with dynamic analysis [1,2,4] and unrepresentative datasets [16]. In general, this approach aggregates the results from the two techniques to create a signature or a program behaviour synthesis. In recent years, we have seen the emergence of hybrid models, which have become widely used. Our approach exploits this development in a specific way: firstly, features are extracted and filtered to provide targeted points of analysis that are investigated in the next step. This filtering, which is based on graph mining, provides a good way to focus the dynamic analysis on the main nodes. During this process, features are extracted and stored as a vector to allow for classification of the program as benign or malware.

Hypothesis: It is possible to reduce the scope of dynamic analysis with filtering based on central node detection, to produce a suitable model for the detection of malware and benign programs.

The evolution of the use of various metrics and approaches has given rise to significant changes in malware detection models (MDMs). The lifetime of models has been improved [17], and they have become more tolerant when faced with new types of malware. In this paper, we focus our attention on these recent developments in MDMs.

Overview

Method Analysis

To perform data extraction in order to classify a program using a Boolean statement (i.e. as benign or malware), we can use two techniques: static and dynamic analysis. A third technique exists, but this is a combination of the two previous ones. In the following, we introduce the method of static analysis.

Static

This technique extracts data from the app package without running it. The app package is initially a compressed file with a dedicated apk extension. All of the assets of the app are present in this archive. To extract the data, this technique decompresses the file and decompiles the program to reach the targeted features [1,9,11]. Although this technique provides good accuracy [1], it has major shortcomings, especially when the app is obfuscated [9], since in this situation, much of the data are not accessible even if multiple decompilers are used during the analysis. To avoid this limitation, some improvements have been made, especially in semantic and graph analysis approaches. In the latter, the model interprets the program structure based on a function call graph or control flow graph [14,18,20]. This evolution constitutes an effective improvement for MDMs, but the accuracy provided by this technique decreases over time. Another shortcoming is the huge datasets generated by the graphs. For an average app size of between 30 to 50 Mo, the average size of these graphs is between 5 and 10Mo for call graphs (CGs) and 120Mo and 2.5Go for control flow graphs (CFGs). The amount of calculation needed to analyse these graphs is significant, and to avoid this, researchers have focused their models on extracts of this graph, and particularly when using CFGs. However, the results of this technique generate more false positives than other models, and are not representative of the global program behaviour. There are many libraries that allow researchers to perform static analysis on Android packages, but in this paper, we use the Androguard Python library (https://github.com/androguard/androguard). When transposed to an Android architecture, static analysis can detect numerous features, but many of them may be corrupted or unrepresentative. This is the first argument supporting our approach to MDMs.

Dynamic

This technique requires an environment (virtual or physical) in which the targeted application is run. During execution, the analysis uses a hook to gather some data [11,12]. Here, again, several types of data can be extracted. To use this technique, an analyst can install or inject a hook function into a suspect app package that focuses on a predefined target list.

The main shortcoming of this technique is the time required to apply it [11]. Whereas a static analysis provides a result in a few seconds, a dynamic analysis takes more time to give results. Another drawback is the fact this technique generates a high proportion of false positives. Despite these limitations, this technique is robust when applied to an obfuscated app, as it focuses its resources on input, output and architectural symbols (like memory addresses) rather than on the obfuscated code. Again, there are many libraries for performing dynamic analysis on an Android package; in our approach, we work with the Frida Python [25]. As can be seen, there is a need to implement a third solution in our approach. In the following, we introduce this last analysis technique.

Hybrid method

These consist of combining dynamic and static analyses in the same process [1,2,4,5]. We could use this technique with different approaches, although researchers typically use a complementarity approach [1,2,4,5] in which the two previous techniques are run separately and the results are aggregated in the same artefact (a vector, matrix or other structure). This combination is relatively complex, as we are gathering a heterogeneous dataset (graph, vector etc.) from multiple levels (system, classes, methods, environment etc.). There is therefore a need to create a data structure into which all data can be combined. Abstraction can be applied to provide a representative and exploitable dataset.

Another approach involves using static analysis to select targeted malicious points that will be analysed by dynamic analysis in the next step. Our work uses this approach to limit the waste of time in the dynamic analysis included in our process.

Malware classification

Malware analysis is performed to gather crucial information such as permission usages and argument distributions, in order to construct some features. These features are usually mined by applying algorithms to classify or cluster all of the applications analysed by the model. To achieve this, researchers use machine learning and/or deep learning algorithms. Machine learning algorithms learn and discover the structure of data from a dataset, and can be exploited in a supervised or unsupervised context. Researchers perform feature extraction to build efficient models to provide the best results in view of the aim of their study. There are many such algorithms, of which the most widely used are AdaBoost, K-means, random forest, Lib-SVM and decision tree. These algorithms have some limitations, such as the amount of data that can be supported.

As an alternative, deep learning algorithms can also used to perform the same goal, but these do not require feature selection. These algorithms mine the dataset to detect the best neural structure to classify the data. Moreover, they can support a dataset with a large size, and their performance improves as the dataset grows. They may be recurrent, artificial, or convolutional. However, these algorithms have several limitations, such as the computing power needed to run them and the minimum size of the dataset required to produce good accuracy.

In this work, we use machine learning algorithms to classify numerical data, as the size of our dataset is most suitable for such algorithms. We introduce our approach in the next section. We use a set of algorithms to evaluate the performance provided by our model, including SVM, random forest, AdaBoost and naïve Bayes. In the following, we give an overview of some recent hybrid MDMs and discuss the results provided by these models.

Related Works

There are numerous MDMs, and we focus our discussion here on recent hybrid models. All of these models implement different approaches. For each model, we describe its process, its logical architecture and its goals. Hybrid MDMs use the approach summarised below:

Approach	Role of static analysis	Possible output
Complementarity approach	Aggregate the results provided by the two methods	Score, signature, or decision

Marvin

Designed by Lindofer et al. [1], this model combines static and dynamic analysis, and was the first type of model to establish a compromise score for an app. This analysis is based on the DroidMat dataset, and the result is in the form of a signature. The model then performs a dynamic analysis and aggregates all of the results to provide a global evaluation of the app. We note that this model uses a complementarity approach. Aggregation provides a global evaluation score that can be used to give advice to the user about app management (e.g. deploying or deleting it), and the user makes the final decision. This model includes the following component blocks in its architecture:

- Static analysis
- Dynamic analysis
- Database
- Parser and extractor for the app package
- Area with specific scripts
- Parser and tracer for the binaries in the app package

Nowadays, this architecture is typical for MDMs. We find the use high-level programming language like Python and Javascript for specific needs. However, its main advantage is the employment of machine learning to perform the classification. Although this model provides a global score based on probabilistic calculation with a good level of accuracy, it is not resilient when faced with an obfuscated app.

Droid ranger [01]

This model [1] performs an analysis via an app previously downloaded from the Google Play Store. It uses a downloading mechanism that is directly connected to the store and based on a similar interface to that used in the apkpure platform [26].The scope of analysis may be large, and the model is adaptable and can provide results quickly, with a good level of accuracy. Another advantage is its interconnection with the main market store for Android applications as mentioned previously. A specific feature of this approach is that static analysis is not automated, but simply assisted. Machine learning techniques are not implemented. This configuration provides a high level of accuracy, but this gain is relative. Recent papers have tended to report good levels of accuracy with full automation of this process, and we will highlight this in our analysis of the next model. Droid Ranger does not compare its results with a signature database or any dataset, and uses a heuristic method to analyse function calls. This method provides good resilience when dealing with new types of malware, as it integrates abstraction into its extracted data. We find this analysis technique in intrusion detection systems.

Finally, this model has a similar architecture to that of the Marvin model.

M0Droid [04]

Designed by Wu et al. [5], this model uses static analysis in the first step to extract the required permissions, native function calls, and API calls present in the app. It also extracts the library imports and class imports mentioned in the code. This allows the model to integrate a complete set of interactions into its scope. Furthermore, it analyses the information present in the app certificate (such as a signature, or author) and uses the well-known Google Bouncer tool to create a trust score for the app. In the next step, it uses dynamic analysis to trace the system and function calls performed during execution of the app. This model also measures the energy consumption for each process directly linked with the app; this metric provides important information, as malware programs use several techniques (such as logic bombs) requiring a high level of energy consumption. The data are registered with the

TaintDroid tool to automate the process. This model has an architecture that is very similar to that of the M0Droid model.

Droidmat [05]

Designed by Wu and al, this model uses static analysis in the first step to extract required permissions, natives' functions calls, and APIs' calls present on the app. Also, it extracts library imports and classes imports mentioned on the code. This allows the model to integrate a complete set of interactions on its scope. Furthermore, it analyses information presents in the app certificate (as signature, author) like the famous tool "Google Bouncer" to make a trust score of the app. In the next step, it uses dynamic analysis to trace system and function calls perform during the app execution. However, this model measures the energy consumption for each process directly linked with the app process. This metrics provides a relevant information because malware programs use several techniques (as logic bomb) requiring a high level of energy consumption. This data is registered with "taintdroid" tool to automate the process. This model has an architecture very similar to the M0Droid model.

Madam [23]

Designed by Saracino et al. [23], this model carries out a multilevel analysis (at the kernel, flow and app levels). Although flow analysis is not a typical method, as we saw for the previous models, the combination of this metric with other features can provide a new perspective for analysis. Furthermore, this model considers the energy consumption, a metric that provides a good way to identify obfuscation techniques in the app. At the dynamic analysis stage, this model extracts several common features (memory area access, function and system calls). These features are correlated with the previous metrics. The static analysis extracts some metadata included in the app package (app certificate, manifest) and other typical data (required permissions, class declarations). This set of data is aggregated with the results of dynamic analysis to classify the behaviour of the app. The static analysis carries out the first level of classification, which is then correlated with the results of dynamic analysis. As we can see, this model uses a complementarity approach.

Bride maid [02]

Designed by Delorenzo et al. [2], this model uses a complementarity approach. In the first step, it uses static analysis to extract several features, and the results are aggregated with those of the second step (dynamic analysis). The architecture of the model is similar to that of the Marvin model, except that this model uses multilevel hooking (local and distant environment): in the local environment, it listens for kernel and Dalvik events, while in the distant environment it listens for data transfers.

Designed by Delorenzo et al. [2], this model uses a complementarity approach. In the first step, it uses static analysis to extract several features, and the results are aggregated with those of the second step (dynamic analysis). The architecture of the model is similar to that of the Marvin model, except that this model uses multilevel hooking (local and distant environment): in the local environment, it listens for kernel and Dalvik events, while in the distant environment it listens for data transfers.

Summary

We can conclude that the main approach used in these models is the complementarity approach; this provides good resilience faced with obfuscated apps, but the lifetime of the model is poor. Another problem is the waste of time involved in performing the dynamic analysis during the process. This problem is well known, and there are currently two strategies for avoiding it. The first consists of paying attention to a few samples of dynamic interactions; however, the drawback of this method resides in the risk of exploiting an unrepresentative dataset for classification. The second strategy involves determining the main suspect target without extracting a huge proportion of dynamic interaction. We believe that the second solution is a good way to build an MDM. In the next section, we introduce our approach to building an MDM, which exploits a representative dataset of app behaviour and is generalisable to any execution context.

In this work, we define a new approach, which we refer to as an exploration approach. This is similar to the complementarity approach except that the results of the static analysis are stored as a vector and can also be used to carry out filtering, which reduces the dynamic analysis required to the most important points. We can define our approach as follows:

Approach	Static analysis role	Possible output
Exploration approach	Carry out targeting to perform local dynamic analysis	Signature (with or without an enrichment mechanism) deep analysis, decision

Methodology and Results

In this section, we introduce the functioning of our model. Firstly, we focus our attention on the architecture of the model, which consists of two stages that handle extraction and classification. For extraction, our model uses an Android application to perform data gathering. This stage includes the following logical steps:

1. We carry out a static analysis, driven by a Python script using the Androguard library, and gather data to build a static vector and the callgraph of the app. The static vector is defined in Table 1.
2. Filtering is performed on the callgraph to extract the critical nodes. These have a high proximity with other call include the graph. These nodes have the following properties:
    - C1: They have a high degree of centrality and proximity (1000 first high score nodes).
    - C2: They manage a minimum of two arguments.
    - C3: They are not Google addmanaging or credential API.
    - C4: They have an in degree and an out degree included in the 0.95 quantile for the twice degree.

    Filtering is performed using the algorithm presented in Table 4.
3. A dynamic analysis is then performed on a virtual machine which uses Android x86 to run the mobile app. During execution of the app, data extraction is performed to extract some information on the mother class concerned by all nodes targeted in the previous step. This extraction is driven by a Python script and the Frida library. We also found a set of hooking functions that were dynamically generated to perform the extraction. At the end of this process, the dynamic vector is created, as defined in Table 1.
4. Finally, the static and dynamic vector are aggregated to form the final vector for the application. This aggregation is implemented using a Python script.

The data extracted from the app are summarised in Table 1 below. When the extraction of data is complete, the model applies the classification module, which consists of three instances: the final app vector built in the previous stage, a dataset, and a neural network classification model. This module is implemented using Python scripts and a set of libraries (sklearn, pandas, numpy). In the following section, we introduce the processing applied at this stage and then describe the fine processing of the dataset and the classification model. Our model applies a neural network and is trained on the dataset. The final vector app is then analysed to predict the type of application (benign or malware).

Tab 1: Extracted data and vector composition

App data	Vector index and type of data	Stage of the model	Vector
Oppacification	Encrypting and renaming: Boolean data	Static analysis	Static
Local components interaction	NbPermissions: total number of permissions used NbPermissionsDangerous: total number of dangerous permissions used NbPermissionsSignature: total number of permissions used that need a user signal NbandroidAPI: Android API number used in the app	Static analysis	Static
Data storage interaction	Nbprovider: number of providers included in the app	Static analysis	Static

	Nbreceiver: number of receivers included in the app		
Outside environment interaction	Nbservices: number of services included in the app	Static analysis	Static
Local artefacts	stringNBGlobal: total number of strings (sum of all following type) UniCodeNb: number of URI strings UrlNb: number of URL strings AffectNb: number of variable instantiations BaliseNb: number of tag strings SpecialNb: number of special character strings CallNb: number of calls ClasseNb: number of classes ChaineNb: number of strings not corresponding to previous types excluding stringNBGlobal	Static analysis	Static
Argument distribution	AD_SYSTEM: number of arguments provided by the OS or the Java language. This metric is extracted for all classes targeted in the filtering step. AD_CUSTOM: number of custom arguments defined by the programmer. This metric is extracted for all classes targeted in the filtering step.	Dynamic analysis	Dynamic vector
Nature of library calls	NLC_INTERN: Percentage of call custom. This metric is extracted for all classes targeted in the filtering step. NLC_EXTERN : Percentage of call system. This metric is extracted for all classes targeted in the filtering step.	Dynamic analysis	Dynamic vector
Average instance size	ASI : Average instance size for each class. This metric is extracted for all classes targeted in the filtering step.	Dynamic analysis	Dynamic vector
Dynamic loaded origins	DLCOI : number of dex used in the app to store the app classes.	Dynamic analysis	Dynamic vector
Number of argument types	NBTYPE : total number of argument types in the app.	Dynamic analysis	Dynamic vector
WMC	Ratio of number method included in class for each class	Dynamic analysis	Dynamic vector
RFC	Average of internal and external method calls in each class. This metric is extracted for all classes targeted in the filtering step.	Dynamic analysis	Dynamic vector
NOC	Average of subclass in each class. This metric is extracted for all classes targeted in the filtering step.	Dynamic analysis	Dynamic vector

In the following, we describe the global functioning of our model, and then define the two crucial components of the classification step. We also note that the dataset was created with a set of malware samples provided by the VirusShare organisation and a random list of benign applications provided by Apkpure. Malware samples were drawn from the period 2012–2018 from those archived on the VirusShare website. To qualify these samples, we sent them via the API services of VirusTotalto determine the most probable type of application. The results were received in XML format, all of the detection results are stored in a global database, and a frequency calculation was applied to all records. For each app, 72 detection results were provided by the analysis engines integrated in the VirusTotal platform. The real type for each record was assumed to be the detection result with the highest frequency.

After, all types are determined the process of our model, previously exposed, is performing an all apps include in our dataset. We found the dataset composition shown in Table 2. As we will show in the next section, the construction of our dataset is quite unusual because our model relies on interdependent steps. At each stage, the result of the previous stage affects the functioning of the following stage. For the two first steps in the extraction process, the probability of being able to perform the analysis is good, and nearly 92% of the Android app can be analysed. However, this does not hold for the dynamic analysis, and especially for the malware samples. These contain numerous ways to evade processing, and our model therefore need certain adjustments to deal with them. After a lot of integration, our model has a probability of being able to perform a dynamic analysis of around 62%. In view of this, we used a large population of malware samples to construct our actual dataset population.

Tab 2: Dataset population

Benign	Rootkit	Trojan	Spyware	Botnet	Ransomware	Total
461	62	289	89	56	115	1072
43.00 %	5,78 %	26,96 %	8,30 %	5,22 %	10,73 %	

Our dataset is used with machine learning algorithms in our classification model. A set of typical algorithms are used to test the detection performance of our model. We discuss this in more detail on the result part (random forest, LibSVM, AdaBoost and naïve Bayes). In the following, we describe the results obtained from applying our model. Firstly, we evaluate our approach using the measures of accuracy, F1-score, recall and precision, which are calculated as shown below.

$Accuracy = \frac{TP + TN}{Total}$	$Recall = \frac{TP}{TP + FN}$
$F1\ Score = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$	$Precision = \frac{TP}{TP + FP}$

TP, TN, FP and FN are the true positive, true negative, false positive and false negative results, respectively, provided by the classification process. These measures are applied because the aim of our model is to produce a classification, and they are frequently used to evaluate the performance of a model in this context.

Table 3: Performance of our model

Algorithms	*Accuracy*	*F1_Score*	*Recall*	*Precision*
Random Forest	0.984	0.955	0.955	0.955
SVM	0.979	0.945	0.955	0.955
AdaBoost	0.930	0.935	0.935	0.935
Naives Bayes	0.966	0.908	0.908	0.908

As we can see from Table 3, our approach provides a precision of close to 0.984% with a random forest algorithm. This result was obtained from a k-fold execution of our model with k = 10. The precision is very similar to that of other hybrid models, and indicates that we can achieve a good level of malware detection with features representative of the global app behaviour in a hybrid model. In our model, we use filtering to limit the scope of the dynamic analysis, and the result of this filtering shows the behaviour of the central calls included in the application.

These results validate our approach and our hybrid model, with a classification based on a Boolean statement. The filtering applied in the second step of our process provides a good level of targeting, as illustrated by the detection accuracy provided. Hence, our main hypothesis is supported.

Conclusion

Through the implementation of a filtering process, we can obtain a representative view of the central nodes included in an Android application. This process provides respectable results in terms of malware detection, based on a simple Boolean statement (benign or malware). However, the operation of our model relies on heavy and restrictive preprocessing when building our dataset. This causes a waste of time and malware samples to perform our analysis before being able to generate classification results.

To avoid this limitation, we have several options. The first and easiest would be to improve the size of our dataset to increase the precision of our model. In this approach, a floor of 10,000 records would be applied to the size of our dataset. Having shown that the data extraction process of our model is robust, we can apply it to an external dataset. We will therefore use the CiCMalDroid dataset in future work. Secondly, to improve the efficiency of the dynamic analysis, we could implement more circumvention methods to deal with the evasion technique implemented in malware. Finally, further investigation is needed into the use of more node filtering, to allow us to integrate into our dataset some measures from these extracted nodes. It is possible that these data could provide insights into the behaviour of the app.

Tab 4: Filtering Algorithm – Pseudo-Code

```
Vars :
        Integer n1, n2, NbArguments, SizeNode
        Float AVGDegreeProximity, AVGDegreeCentrality, DegreeCentrality, DegreeProximity, InDegree, OutDegree, ProximityQuantile, CentralityQuantile, TabDegreeProximity[n2], TabDegreeCentrality[n2], x
        Character Methods, Nodes, TypeNode, MethodsReg
MatrixFiltered[n1][9] FilteringNode(MatrixBrut[n2][9])
Begin
        Integer i
        For i = 1 to n2 Do
        TabDegreeCentrality[i] <- MatrixBrut[3][i]
        TabDegreeProximity [i] <- MatrixBrut[6][i]
End

x = 0,95
ProximityQuantile <- Quantile(x, TabDegreeProximity)
CentralityQuantile <- Quantile(x, TabDegreeCentrality)

Begin
        Integer i
        For i = 1 To n2 Do

                Methods <- MatrixBrut [2][i]
                Nodes <- MatrixBrut [1][i]
                NbArguments <- MatrixBrut [9][i]
                SizeNode <- MatrixBrut [8][i]
                DegreeProximity <- MatrixBrut [6][i]
                DegreeCentrality <- MatrixBrut [3][i]
                OutDegree <- MatrixBrut [4][i]
                InDegree <- MatrixBrut [5][i]
                TypeNode<- MatrixBrut [7][i]
                If DegreeProximity > ProximityQuantile and DegreeCentrality > CentralityQuantile and Methods contains MethodsReg Then
                        MatrixFiltered (i) <- [Nodes, Methods, DegreeCentrality, OutDegree, InDegree, DegreeProximity, TypeNode, SizeNode, NbArguments]
                End If
        End For

End
```

Reference

[01] LINDORFER, Martina, NEUGSCHWANDTNER, Matthias, et PLATZER,Christian. Marvin: Efficient and comprehensive mobile app classification throughstatic and dynamic analysis. In: 2015 IEEE 39th annual computer software andapplications conference. IEEE, 2015. p. 422-433.

[02] DE LORENZO, Andrea, MARTINELLI, Fabio, MEDVET, Eric, et al.Visualizing the outcome of dynamic analysis of Android malware with VizMal.Journal of Information Security and Applications, 2020, vol. 50, p. 102423.

[03] https://github.com/Dam4323/EGDataset

[04] DAMSHENAS, Mohsen, DEHGHANTANHA, Ali, CHOO, Kim-KwangRaymond, et al-. M0droid: An android behavioral-based malware detectionmodel. Journal of Information Privacy and Security, 2015, vol. 11, no 3, p.141-157.

[05] WU, Dong-Jie, MAO, Ching-Hao, WEI, Te-En, et al. Droidmat: An-droid malware detection through manifest and api calls tracing. In: 2012 Seventh Asia Joint Conference on Information Security. IEEE, 2012. p. 62-69.

[06] KARBAB, ElMouatez Billah, DEBBABI, Mourad, DERHAB, Abde-louahid, et al. MalDozer: Automatic framework for android malware detectionusing deep learning. Digital Investigation, 2018, vol. 24, p. S48-S59.

[07] SUAREZ-TANGIL, Guillermo, DASH, Santanu Kumar, AHMADI, Man-sour, et al. Droidsieve: Fast and accurate classification of obfuscated androidmalware. In: Proceedings of the Seventh ACM on Conference on Data andApplication Security and Privacy. 2017. p. 309-320.

[08] AONZO, Simone, GEORGIU, Gabriel Claudiu, VERDERAME, Luca,et al. Obfuscapk: An open-

source black-box obfuscation tool for Android apps.SoftwareX, 2020, vol. 11, p. 100403.

[08] YERIMA, Suleiman Y., SEZER, Sakir, MCWILLIAMS, Gavin, et al.A new android malware detection approach using bayesian classification. In:2013 IEEE 27th international conference on advanced information networkingand applications (AINA). IEEE, 2013. p. 121-128.

[09] SUAREZ-TANGIL, Guillermo, DASH, Santanu Kumar, AHMADI, Man-sour, et al. Droidsieve: Fast and accurate classification of obfuscated androidmalware. In: Proceedings of the Seventh ACM on Conference on Data andApplication Security and Privacy. 2017. p. 309-32022

[10] ALZAYLAEE, Mohammed K., YERIMA, Suleiman Y., et SEZER,Sakir. DynaLog: An automated dynamic analysis framework for characterizingandroid applications. In: 2016 International Conference On Cyber Security AndProtection Of Digital Services (Cyber Security). IEEE, 2016. p. 1-8.

[11] SCHMIDT, A.-D., BYE, Rainer, SCHMIDT, H.-G., et al. Static anal-ysis of executables for collaborative malware detection on Android. In: 2009IEEE International Conference on Communications. IEEE, 2009. p. 1-5.

[12] KANG, BooJoong, YERIMA, Suleiman Y., MCLAUGHLIN, Kieran, etal. N-opcode analysis for android malware classification and categorization. In:2016 International conference on cyber security and protection of digital services(cyber security). IEEE, 2016. p. 1-7.

[13] SHABTAI, Asaf, TENENBOIM-CHEKINA, Lena, MIMRAN, Dudu, etal. Mobile malware detection through analysis of deviations in application net-work behavior. Computers Security, 2014, vol. 43, p. 1-18.

[14] JANG, Jae-wook, WOO, Jiyoung, YUN, Jaesung, et al. Mal-netminer:malware classification based on social network analysis of call graph. In: Pro-ceedings of the 23rd International Conference on World Wide Web. 2014. p.731-734.

[15] FENG, Yu, ANAND, Saswat, DILLIG, Isil, et al. Apposcopy: Semantics-based detection of android malware through static analysis. In: Proceedings ofthe 22nd ACM SIGSOFT international symposium on foundations of softwareengineering. 2014. p. 576-587.

[16] ALSMADI, Tibra et ALQUDAH, Nour. A Survey on malware detec-tion techniques. In: 2021 International Conference on Information Technology(ICIT). IEEE, 2021. p. 371-376.

[17] WU, Yueming, LI, Xiaodi, ZOU, Deqing, et al. MalScan: Fast market-wide mobile malware scanning by social-network centrality analysis. In: 201934th IEEE/ACM International Conference on Automated Software Engineering(ASE). IEEE, 2019. p. 139-150.

[18] YANG, Yang, DU, Xuehui, YANG, Zhi, et al. Android Malware De-tection Based on Structural Features of the Function Call Graph. Electronics,2021, vol. 10, no 2, p. 186.

[19] KINABLE, Joris et KOSTAKIS, Orestis. Malware classification basedon call graph clustering. Journal in computer virology, 2011, vol. 7, no 4, p.233-245.23

[20] XU, Ming, WU, Lingfei, QI, Shuhui, et al. A similarity metric methodof obfuscated malware using function-call graph. Journal of Computer Virologyand Hacking Techniques, 2013, vol. 9, no 1, p. 35-47.

[21] YERIMA, Suleiman Y., SEZER, Sakir, MCWILLIAMS, Gavin, et al.A new android malware detection approach using bayesian classification. In:2013 IEEE 27th international conference on advanced information networkingand applications (AINA). IEEE, 2013. p. 121-128.

[22] RASCAGNERES Paul. Securite informatique et malwares: analyse desmenaces et mise en œuvre des contre-mesures. St Herblain : Editions ENI, 2019.

[23] SARACINO, Andrea, SGANDURRA, Daniele, DINI, Gianluca, et al.Madam: Effective and efficient behavior-based android malware detection andprevention. IEEE Transactions on Dependable and Secure Computing, 2016,vol. 15, no 1, p. 83-97

[24]https://github.com/androguard/androguard

[25] https://frida.re/

[26] https://apkpure.com

List of abbreviation

MDM : malware detection model

CG : Call graph

CFG : Control flow graph