Journal of Electronic Research and Application, 2025, Volume 9, Issue 4
http://ojs.bbwpublisher.com/index.php/JERA
ISSN Online: 2208-3510
ISSN Print: 2208-3502

Genetic Analysis of Android Malware

Linna Wang*

Tianjin University, Tianjin 300072, China

**Author to whom correspondence should be addressed.*

Abstract: With the proliferation of Android malware, the issue of traceability in malware analysis has emerged as a significant problem that requires exploration. By establishing links between newly discovered, unreported malware and prior knowledge from existing malware data pools, security analysts can gain a better understanding of the evolution process of malware and its underlying reasons. However, in real-world scenarios, analyzing the traceability of malware can be complex and time-consuming due to the large volume of existing malware data, requiring extensive manual analysis. Furthermore, the results obtained from such analysis often lack explanation. Therefore, there is a pressing need to develop a comprehensive automated malware tracking system that can provide detailed insights into the tracking and evolution process of malware and offer strong explanatory capabilities. In this paper, we propose a knowledge graph-based approach that uses partial API call graphs comprising semantic and behavioral features to reveal the traceability relations among malware and provide explainable results for these relations. Our approach is implemented on a dataset of over 20,000 malware samples labeled with family information, spanning a time period of 10 years. To address the challenges associated with the complexity of analysis, we leverage prior knowledge from existing malware research and a branch pruning method on call graphs to reduce computational complexity and enhance the precision of explanations when determining traceability relations.

Keywords: Malware gene traceability; Malware analysis; Android

Online publication: August 8, 2025

1. Introduction

Since 2007, Android has been the most widely used mobile operating system [1]. IDC statistics predict that by 2023, Android mobile phones will have a market share of 87.1%, amounting to 1.519 billion units. However, the increasing popularity of Android has led to more users falling victim to Android malware. The openness and freedom of Android application development have made it easier for attackers to create malware with greater impact.

Malware creators are constantly working to evade detection by security engines, leading to the continuous evolution of malware. To truly understand the ecological development of malware and improve defense measures, it is essential to trace malware behavior. Malware traceability is mainly based on its malicious behavior [2] and

can be achieved by exploring the homology of malware codes. Currently, malware analysis relies on various methods, such as static or dynamic program analysis [3], machine or deep learning-based classifiers [4]. These methods provide security experts with effective solutions but still heavily depend on experts with a strong security background. Comparing malware in the existing large and complex sample data is time-consuming. The traceability results lack sufficient historical data support, making it unclear about the evolutionary chain and source of the samples. The interpretability of the traceability results is weak, leading to a lack of effective measures to deal with attacks. Therefore, a complete automated malware traceability system is necessary, which can depict malware traceability and evolution in detail with strong explanatory power.

2. Preliminary experiment

Since we want to conduct malware evolution analysis, a large-scale malware dataset, preferably with timestamps and a long time span, is necessary. Likewise, the malware should be labeled with family information so that we can distinguish the samples within an initial setup in the family dimension. To achieve this, we conducted two pre-experiments.

Data collection and annotation: We list the key selection criteria for collecting the malware data as follows:

(1) To analyze the evolution of Android malware on the time axis, the samples need to be distributed evenly in the past ten years (e.g., from January 2011 to December 2021).

(2) To use the family classification tool to add family labels to the samples, the samples need to have been verified by VirusTotal.

(3) To make the family label of malware more credible, the samples need to have been detected with more than 10 viruses.

We select the samples in Androzoo based on the above three criteria. Since the selection of samples under this standard is random, sample bias may have an impact on the number of families and the family distribution. Ruling out these possible effects on our experimental results is needed. Considering the limited storage space and the practical demands, we download 2,000 samples per year.

3. Approach

The framework comprises three modules: malware representation, coarse-grained traceability analysis, and fine-grained traceability explanation. The first module uses two types of features to characterize malware, semantic information, and malicious behavior. These modules conduct traceability analysis between families and samples. The third module explains traceability at the behavior level along with malware genetic methodology.

3.1. Family-level graph

In this section, we introduce the first module of the framework, which can abstract a subset of malware (e.g., entire family of our collected dataset) in a feature vector and build a malware family graph to explore the relations among families at the coarse-grained level.

In order to build a comprehensive and accurate malware family knowledge graph, we extract features, including 60 permissions and 95 APIs by Chen *et al.* and the family classification results of malware to characterize the family and the malware in the family. The vectorized representation of features is to abstract the relation of triples in the malicious family knowledge graph as text semantics. For each family, the features

contained in each sample belonging to the family are mapped to the joint vector space of the feature dictionary. Each dimension in the vector represents the occurrences of a feature in the family. If a given sample in a family contains a feature in the dictionary, then the value corresponding to the feature in the vector matrix is incremented by 1. To distinguish the importance of different features, we use the theorem of TF-IDF to calculate and assign weights.

Through the above steps, we can get the vectors of which each represents a family, and the weights of all features in the corresponding family. We then calculate the cosine distance between any two families.

By observing the final result of genetic analysis, when we choose 0.7 as the threshold, we can not only accurately capture the families with evolutionary relations, but also obtain some data for our analysis. When the distance between the two families exceeds 0.7, we identify that there is a genetic relation between the two paired families, and this relation is reflected in our family graph as a path. Finally, we can establish a derivation knowledge graph at the coarse-grained family level.

3.2. Sample-level graph

In order to empower the capability of the knowledge graph to investigate the precise relation between different malware samples, the second module, which is a deep learning-based hierarchical classifier, is designed for refining the granularity of the knowledge graph from family to sample level. Existing malware family classifications only focus on the features of families. In other words, they ignore the impact of time changes on the implementation of malware belonging to the same family, but in fact, these impacts are widespread and important to evolution analysis, such as changes in API definition and calling mechanisms. Hence, we need a new classifier that can do classifications in two dimensions, i.e., the family feature and the release year simultaneously. To this end, we establish a hierarchical structure to fuse the family and year tags and set up an encoder to perceive the relation between tags. By adopting a multi-label attention mechanism, the final input vector is generated from the encoded family feature and malware sample feature vectors and fed into a CNN to train the classifier. An overview of the final input generation is shown in **Figure 1**. The hierarchical structure of family types with year numbers is defined as a directed diagram G = (V,E), whose rough structure is shown in the dashed box "Structure Encoder" in **Figure 1**.

Since we want to take both the prior knowledge (i.e., family and year information) and key implementation features as the used features in classification, an attention-aware deep neural network is applied to perceive the final vector representation of the overall sample information. We take the encoded family-year vector and the sample vector as inputs, and output a final vector representation, we map the final features to low-dimensional vectors and input them into CNN. For every family, two-thirds of the malware is used as the training data, and the rest is used as the test set. With the pre-trained model, we can get the candidate families of a given sample and the probabilities that represent the sample belonging to the families.

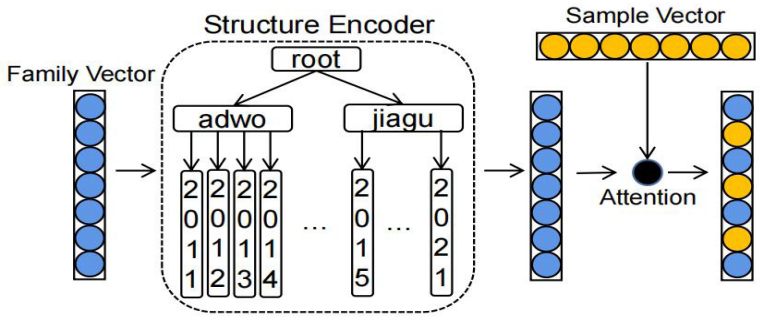

Figure 1. Hierarchical feature structure

3.3. Behavior-level graph

After knowing the evolution direction of every single malware sample, we aim to investigate the root cause of the high similarity between samples, such as code reuse. Therefore, we analyze the relation between samples at the behavioral level by processing and analyzing their function call graph, and cluster similar malicious behaviors among different samples to enable genetic analysis at a more fine-grained level. We also define the key implementation features of the behaviors as malware genes to enhance the explainability of evolution analysis and the usability for mitigation. No matter whether the malware sample is judged to belong to its original family or not, there must be some similar behaviors between it and the samples of the judged family. For example, when they want to implement a behavior, they must call an API with specific parameters. These APIs may all be the same, but the parameters of these APIs are not necessarily the same, and the methods of being called are not necessarily the same. From this point of view, it is not enough to distinguish the behavior of malware only by calling the sensitive API, and other elements, including API call parameters and other APIs that are called or called need to be considered. This is also the reason why we do not use discrete API sequences directly. The call subgraph with sensitive API can outline the behavior information of malware more completely. Before diving into the details, we want to explain some key terms first in order to facilitate the introduction of our work later.

(1) CG: Call Graph

(2) SCG: Sensitive Call Graph

(3) SSG: Sensitive Subgraph

CG contains the function call relation of the entire Android application software, which can describe the features of the malware in great detail. However, the CG is cumbersome to process. Considering that the size of user data nearly reaches 2 TB, directly using CG will cost enormous time and computing space to analyze malware. Although sensitive APIs represent only a small portion of the total APIs, most malware can only access sensitive information or perform malicious tasks through them, according to the design of the Android OS. Since our target is to reduce the complexity of CG and retain as much implementation related to malicious behaviors as possible, we then highlight the sensitive APIs related to malicious behaviors in the CG and name the obtained new graph as the sensitive function call graph [5]. Totally, we use a set of 9,730 sensitive APIs summarized by Fan *et al.* We define the sensitive function call graph (SCG = (V′,E′)) as a maximum subgraph of the function call graph (CG = (V,E)) containing all sensitive API nodes and their parent nodes.

After processing, each sample is represented as an SCG. It is found by observation that most of the subgraphs have different structures, whereas there are some similar subgraphs among the SCGs of different samples. Matching two SCGs directly is very inefficient, mainly because determining a problem of subgraph isomorphism actually belongs to the classic NP-complete problems [6]. Therefore, in order to accurately locate similar behavior between two different samples and reduce the computational complexity, we divide the SCG into a set of smaller subgraphs. We choose infomap as the community partition algorithm because more subgraphs are obtained compared to other algorithms, and the number of nodes in each subgraph is relatively small, which leads to a lower complexity of similarity calculation [7]. Most of the subgraphs generated by the community division are irrelevant to sensitive data, and these subgraphs are of little help in extracting malicious behaviors of malware. Therefore, we keep subgraphs containing at least one sensitive API node and define them as SSGs. SSGs also have their corresponding weights w(SSG), representing the SSG's importance relative to the family.

Because malicious behavior is mainly reflected by sensitive APIs, we consider sensitive API nodes of SSG as

the key factor when calculating SSGs similarity. The algorithm mainly depends on the structure equivalent of the sensitive API nodes; we exploit the topological structure of the sensitive APIs to calculate the similarity between two SSGs.

Define N clusters, where N is the number of SSGs of the test sample. Each cluster contains only one SSG from the test sample. By traversing all SSGs of the samples in the candidate families, the similarity between the traversed SSGs and each SSG of the test sample in the clusters is calculated. Referring to the calculated data of SSGs similarity and our manual verification, we choose a threshold of 0.75 to better cluster SSGs that can represent the same malicious behavior. If the similarity is higher than the threshold, which is 0.75, the traversed SSG is added to the cluster. If multiple clusters satisfy the SSG addition condition, we select the cluster with the highest average similarity. Finally, the subgraphs that express malicious behaviors, which are similar to those of the test sample, are obtained according to the results of clustering.

The biological definition of a gene is the basic genetic unit that controls biological traits, and the traits of malware are reflected in the unique malicious behavior exhibited by the malware [8]. For malware, its gene can be understood as a partially abstracted CG of malware, which is the smallest functional unit that inherits malicious behavior and the primary genetic unit that controls the malicious function of malware. It is a relatively independent unit of genetic information that can be recombined in different modules in various ways and inherited to future generations. The study of malware genes helps to analyze the function of malware, and also has a certain effect on the homology determination of malware [9,10]. The malware gene in this paper is defined as the SSG possessed by malware.

4. Case study

In this section, we make a genetic analysis through a knowledge graph to investigate the behavior changes in "jiagu" and "adwo" and to expose the main reasons for evolution. Due to the space limitation, **Figure 2** illustrates only "jiagu" 2020, "adwo" 2014, and its relevant families, including "dnotua" (2017, 2019, 2021), "youmi" (2015, 2017), and "hiddenad" (2020). Each family has 248 malware on average, while "jiagu" 2020 is the largest family with 1198 malware, but the least one, "dnotua" 2021, contains just 15 malware. The relations between "jiagu" 2020 and the other three families are declared next.

4.1. Relation with "dnotua"—"jiagu"

Apart from the legacy from 2017 to 2021 that implements unauthorized database changes (by loading APIs in classes like "SQLiteDatabase" and "CursorWindow"), family "dnotua" in the year 2019 delivers part of its genes to "jiagu" 2020. As is shown by the yellow lines in **Figure 2**, the behavior manipulates malware to load classes into JVM (Java Virtual Machine) via API "ClassLoader.loadclass" and "Class.getDeclaredField", etc. Even if it seems to be a safe operation, the other APIs ("FileOutputStream.write") in the same gene could be activated to edit local files, which may damage privacy.

4.2. Relation with "youmi"—"jiagu"

As early as 2015, "youmi," which was on the verge of extinction, had a tendency to switch to "jiagu" family in 2016 (purple line in **Figure 2**). At that time, "jiagu" family was still in a precarious position, and it learned from "youmi" family's behavior of popping up advertisements without permission. The banished family "youmi"

successfully inserts its genes (the green lines in **Figure 2**) into "jiagu" 2020. This notes that malware in "jiagu" 2020 gains the capability to open Asset files and extract figures by using "AssetManager.open" and "Resources. getAssets". At the same time, it has malware with harmful functional units that add a malicious payload into the sandbox through "FileOutputStream.write" and its correlated APIs.

4.3. Relation with "hiddenad"—"jiagu"

At the observation of "hiddenad" in the year 2020, we are concerned that its gene sticking into "jiagu" 2020 depicts a representation of achieving pictures from Asset files as similar as "youmi." In **Figure 2**, it stands for a mixture of genes from two malware families, which produces a hybrid malware (e.g., the little circle with two colors) in "jiagu." Additionally, as both of those two families were built in 2020, we manually check at a more specific time, such as the package time, of their malware for evolution reasoning. The empirical result proves a factor that 57% of malware in "hiddenad" 2020, which was created earlier than that in "jiagu" 2020, determines the evolution mentioned above.

The genetic knowledge graph certifies close relations between "jiagu" and other families. We further locate certain malware in "jiagu" adopting similar behaviors compared to malware in other ones, according to the result of the sample-level classification. That similarity, by using the perspective of the behavior level discussion, has incorporated many genes through directly receiving the basic operational elements from another family "dnotua," or copying similar functions merged by two different families, "youmi" and "hiddenad."

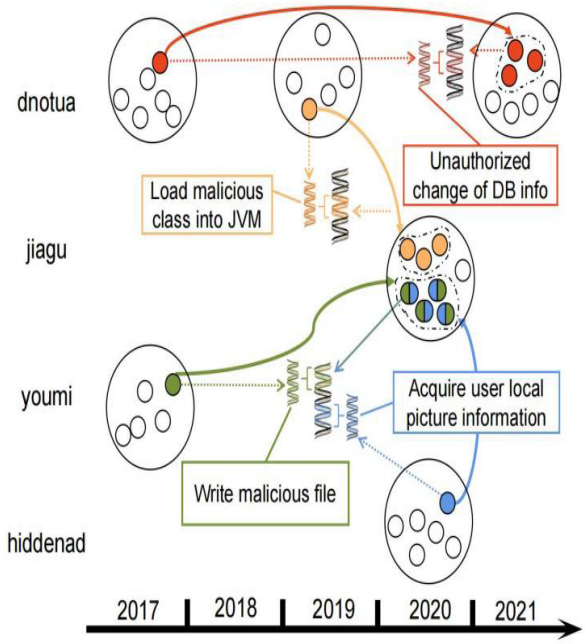

Figure 2. Genetic knowledge graph of "jiagu" and "adwo"

5. Conclusion and future work

This work aims to handle the problems in the evolution analysis of Android malware according to the concept of family-malware-gene. We propose a malware genetic analysis framework hybridizing program analysis technologies and ML/DL methods. To evaluate the effectiveness, a multi-grained knowledge graph based on

20,142 malware is constructed to demonstrate the practical capability. In the future, we will go a step further to automatically reason possible malware developments and newly appearing behaviors, based on the proposed knowledge graph and more Android malware collected with accurate family labels.

Disclosure statement

The author declares no conflict of interest.

References

[1] Willems C, Holz T, Freiling F, 2007, Toward Automated Dynamic Malware Analysis Using Cwsandbox, IEEE Security & Privacy, 5(2): 32–39.

[2] Arp D, Spreitzenbarth M, Hubner M, et al., 2014, Drebin: Effective and Explainable Detection of Android Malware in Your Pocket, in ISOC Network and Distributed System Security Symposium (NDSS), (14), 23–26.

[3] Rastogi V, Chen Y, Jiang X, 2013, Droidchameleon: Evaluating Android Anti-Malware Against Transformation Attacks, in Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, 329–334.

[4] Kolosnjaji B, Zarras A, Webster G, et al., 2016, Deep Learning for Classification of Malware System Call Sequences, in Australasian Joint Conference on Artificial Intelligence, Springer, 137–149.

[5] David OE, Netanyahu NS, 2015, Deepsign: Deep Learning for Automatic Malware Signature Generation and Classification, 2015 International Joint Conference on Neural Networks (IJCNN), IEEE, 1–8.

[6] Ahmadi M, Ulyanov D, Semenov S, et al., 2016, Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification, in Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy (CODASP), 183–194.

[7] Meng G, Feng R, Bai G, et al., 2018, Droidecho: An In-Depth Dissection of Malicious Behaviors in Android Applications. Cybersecurity, 1(1): 1–17.

[8] Chen J, Wang C, Zhao Z, et al., 2017, Uncovering the Face of Android Ransomware: Characterization and Real-Time Detection. IEEE Trans Actions on Information Forensics and Security (TIFS), 13(5): 1286–1300.

[9] Li L, Gao J, Kong P, et al., 2020, Knowledgezooclient: Constructing Knowledge Graph for Android, in 2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), IEEE, 73–78.

[10] Tam K, Feizollah A, Anuar NB, et al., 2017, The Evolution of Android Malware and Android Analysis Techniques. ACM Computing Surveys (CSUR), 49(4): 1–41.