RESEARCH ARTICLE

RSP Science Hub

International Research Journal on Advanced Science Hub
2582-4376
www.rspsciencehub.com
Vol. 07, Issue 07 July

Check for updates

http://dx.doi.org/10.47392/IRJASH.2025.077

Shap Based -Android Malware Detection Using Ensemble Learning

Dr N Anitha Devi[1], C Karthika[2], V Pradeepa[3], C Sharmila[4]

[1]Assistant professor, Dept. of IT, Coimbatore Institute of Technology, Coimbatore, Tamil Nadu, India

[2,3,4]UG Scholar, Dept. of IT, Coimbatore Institute of Technology, Coimbatore, Tamil Nadu, India

Emails: *anithadevi@cit.edu.in[1], 71762207021@cit.edu.in[2], 71762207033@cit.edu.in[3], 71762207047@cit.edu.in[4]*

Article history

Received: 07 June 2025
Accepted: 21 June 2025
Published: 25 July 2025

Keywords:

Android malware detection, Sensitive Function Call Graph, NetworkX, Word2Vec, Smali code, API semantic analysis, shap interpreter, social network analysis.

Abstract

Android malware remains a critical threat to mobile security, demanding robust and transparent detection mechanisms. This approach proposes a complete method to identify malicious Android apps by using code analysis and graph-based techniques, enabling the identification to be more precise and interpretable. The workflow starts with a detailed pre-processing stage, during which APK samples are decompiled. With the help of Baksmali, we retrieve DEX files and decompile them into Smali code, extracting the program behaviour and program flow. Moreover, Androguard is used to retrieve abstract metadata and permission specifications, helping with code semantics inspection. We then build Sensitive Function Call Graphs (SFCGs) for all Android apps, where vertices are sensitive API-calling functions and edges are their calls between functions. We enrich the graphs with both layout-based features, like degree centrality, closeness centrality, and clustering coefficients, and permission patterns in Smali code. Semantic features are extracted by transforming smali code and using word2Vec.The features are then utilized to construct a strong ensemble learning system of multiple individual classifiers. Furthermore, in our effort to further make our detection system more transparent and strong, we employ SHAP to provide model explanations, resulting in attribute-specific explanations for malware classification results. Experiments with a large reference dataset illustrate the performance of the proposed approach towards obtaining accurate, interpretable, and scalable Android malware detection with approximately 99.9%. The system not only adds to security but also promotes transparency, which is crucial in security-critical applications.

1. Introduction

Android cell phones have also become susceptible to attacks due to their popularity and openness since they have an open-source architecture. Holding more than 70% market share worldwide as of March 2025 (StatCounter Global Stats), Android also holds the market for mobile operating systems. Owing to its user and developer community, however, it has also become one of the largest targets for attacks from malicious entities. These take advantage of open system vulnerabilities, interfere with device functionality, and steal users' data. Heuristic and signature-based detection finds it difficult to detect sophisticated forms of threats in the form of malware that hides

behind or morphs through polymorphic transformations. Sophisticated detection techniques to scan structural as well as behavioral patterns to detect known and unknown malware threats bring an urgent need as the scenario is changing by the minute. It is more crucial now than ever before to possess strong, advanced malware detectors since mobile telephones are about to become essential for everyday activities like social networking, online banking, and computer games.

Traditional methods for detecting Android malware can be divided into two primary categories: static analysis and dynamic analysis. Dynamic analysis monitors the behavior of apps while they are operating, but it requires emulation or real device execution, which is time-consuming and resource-intensive. However, static analysis provides a faster and less costly method of examining the code or structure of the program without actually running it. Most modern static analysis methods use machine learning algorithms to extract information like permissions, API calls, or control flow in order to detect malware. But these approaches are usually plagued by two primary issues: (1) excessive dependence on a single type of feature, and (2) an application behavior without semantic knowledge or structural context. Baksmali is initially employed to decompile Smali code for APK samples. This enables detailed analysis of functional behavior and offers a low-level description of the application's logic. Based on this, we build Sensitive Function Call Graphs (SFCGs) so that nodes are functions that call sensitive APIs and edges denote their interactions. Following collection, the Word2Vec model is used to convert the Smali code into semantic embeddings that capture the contextual relationships between methods and instructions. By combining the prediction capabilities of many models, our ensemble learning method for malware classification lowers the risk of overfitting and enhances generalization to unknown infections. Furthermore, we use SHAP (SHapley Additive exPlanations) to provide per feature contribution scores, which increase trust and transparency by allowing security researchers to comprehend the reasoning behind a negative flag for an application.

2. Related Work

Android malware detection has gone from fundamental static or dynamic analysis to advanced machine learning and graph based techniques. The reason for this evolution is not only the need for accurate, explainable, and robust detection systems against threats like code obfuscation and continuous changes in malware, but belonging to the field of information/discovery. Many prior works have proven that graph representations can be instrumental in modeling app behaviors. For example, Gong et al. (2024) presented a method that models behavioral chains by connecting sensitive API calls through Abstract Syntax Tree (AST) structures. Although they used AST structures, we are using sensitive behavior directly at the Smali code level with Sensitive Function Call Graphs (SFCGs), which provide a lower-level and closer to running representation of the app's behavior. Onwuzurike et al. (2019) and Anand et al. (2025) are examples of leading works on static analysis that have paved the way for extracting features like permissions, intents, and API usage. Our work builds on this due to embedding semantic vectors into cleaned Smali code using Word2Vec, and enhancing the feature set using permission-based flags and SFCG-derived structural features—providing a deeper context and more behavioral information. Anand et al. (2025) also established the value of Smali-level semantic features, and were able to develop a deep learning model that utilized a token-level view of the Smali code to successfully classify malware. This provides further justification to use both structural and semantic inputs from the Smali code. Ensemble-based methods have outperformed individual models in classification. Cui et al. (2023) showed that better generalization results from stacking classifiers according to permissions and APIs. Similarly, Nethala et al. (2025) presented a deep ensemble framework that successfully handles unbalanced Android malware datasets by combining neural networks with conventional classifiers. These results are in line with our use of an ensemble that combines Random Forest and XGBoost with soft voting, providing improved robustness and resistance to overfitting. Ensemble-based methods have demonstrated superior performance in classification. The lack of interpretability of the model is a significant gap in previous work. Even though many systems are very accurate, their decision-making process is not very transparent.

We integrate SHAP explainability into our system, drawing inspiration from Sharma et al. (2024), who used SHAP to highlight significant features in malware predictions. This enhancement makes our model more reliable and useful for real-world implementation by allowing it to explain its predictions—by emphasizing which structural or semantic features influenced the classification.

3. Proposed Methodology

Our proposed methodology consists of five stages, viz., pre-processing, SFCG generation, smali code transformation, ensemble model classification, and SHAP-based interpretation, to address the limitations of conventional detection techniques.

3.1. Pre-processing

In this phase, in order to extract classes.dex files from Android APK files, we perform reverse engineering using the tool androguard; subsequently, we extract the smali code of every APK from the obtained DEX file using baksmali-2.5.2, a disassembler. These smali code are organized and stored in hierarchy manner and can be used for further processing.

3.1.1. APK File Structure

An Android APK (Android Package) is essentially a ZIP archive that contains all the components required to install and run an Android application. Once extracted or disassembled, an APK typically includes the following files and directories:

- AndroidManifest.xml: Declares essential information about the app, such as package name, components (activities, services), requested permissions, and more.
- classes.dex: Contains the Dalvik Executable bytecode that runs on the Android Runtime (ART). These files are later converted into smali code during disassembly.
- res/: A directory containing resources such as layouts (.xml), images (.png), and strings.
- lib/: Contains compiled native code libraries for various CPU architectures (e.g., armeabi-v7a, x86).
- assets/: Raw asset files bundled with the app.
- META-INF/: Contains metadata about the APK, including signatures and certificates.
- resources.arsc: Contains precompiled resources

3.2. Sensitive Function Call Graph (SFCG)Generation

This phase constructs SFCG where

3.2.1. Sensitive Node Identification

Sensitive nodes of each apk is identified by comparing it with the predefined sensitive api list(Camodroid) like "Landroid/accounts/ Account Authenticator Activity;->unbindService","Landroid/app/Activity;->clearWallpaper",

3.2.2. Ancestor Tracing

For every sensitive node identified, the control flow is traced backward through the call graph to locate all possible ancestor methods that eventually invoke the sensitive API. This makes sure that context to sensitive behaviour is preserved. We eliminate branches irrelevant to sensitive Figure 1 shows Architecture of Proposed Work
behavior in the Function Call Graph (FCG), thus reducing scope to chains of sensitive behavior alone, reducing computational complexity without sacrificing analytical depth.

3.2.3. SFCG Construction and Storage

A trimmed version of the whole call graph is generated using the networkx Python package, displaying just the nodes involved in sensitive chains. The SFCG is stored in structured JSON form in three main components:
"sensitive_nodes" – a list of API calls recognized as sensitive,
"ancestors" – all approaches that end with the invocation of the sensitive APIs,
"structural_features" – graph measures for each method.

3.2.4. Structural Feature Extraction

For each node of the formed SFCG, we compute a set of social network analysis metrics from network : Degree Centrality ,In-degree and Out degree,Closeness Centrality ,Betweeness Centrality, Harmonic Centrality, PageRank , Clustering Coefficient and Square Clustering Coefficient .These metrics numerically reflect the position and impact of a function within the behavior chain to aid in the extraction of dense structural feature vectors per APK.

3.2.5. Integration of Permission based features

In addition to graph features, we also extract

permission-based features from both the APK manifest and the Smali code.

These include flags such as

- uses_SEND_SMS, uses_READ_SMS, uses_RECEIVE_SMS
- uses_READ_PHONE_STATE, uses_CAMERA, uses_RECORD_AUDIO
- uses_INTERNET, uses_ACCESS_FINE_LOCATION, etc.

These features are binary indicators, representing the presence of potentially risky API usages. Figure 2 shows APK File Structure

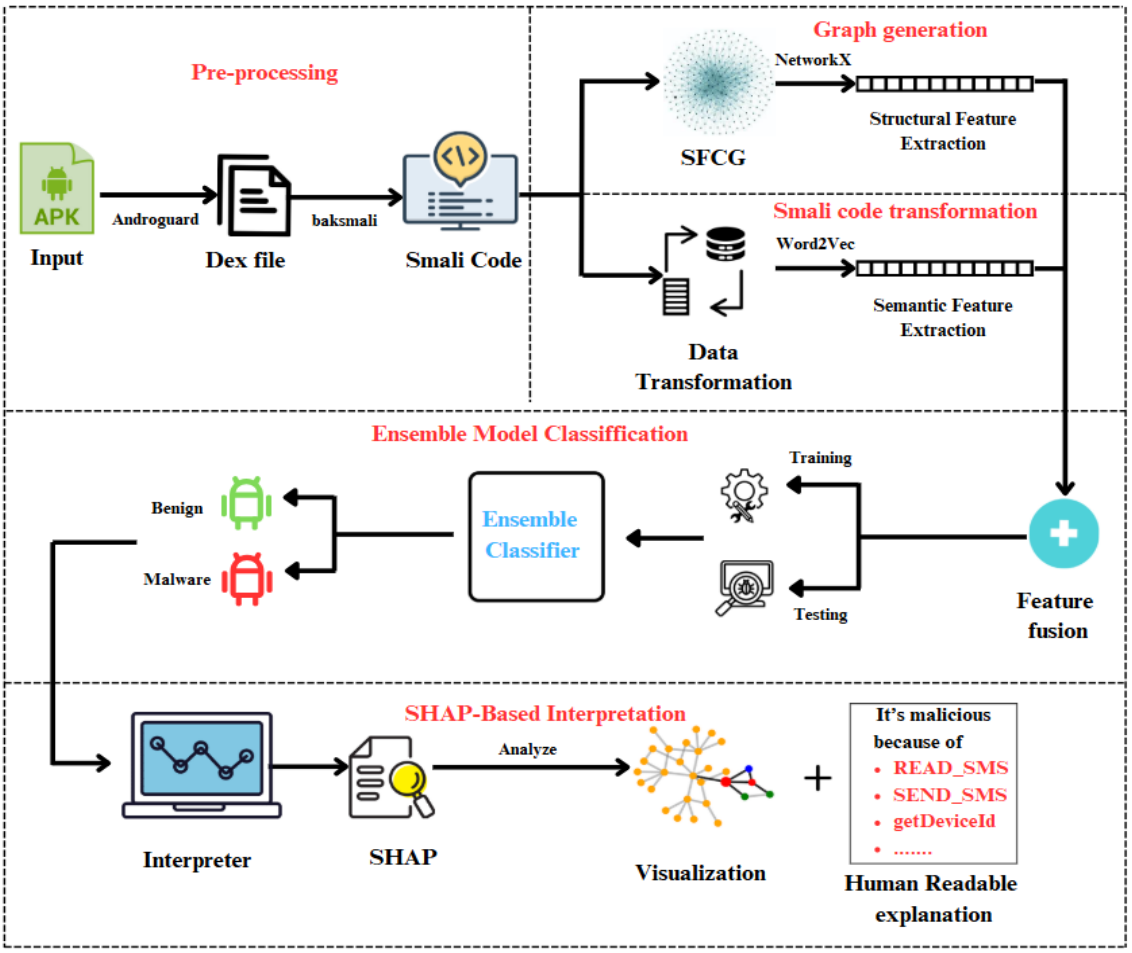

Figure 1 Architecture of Proposed Work

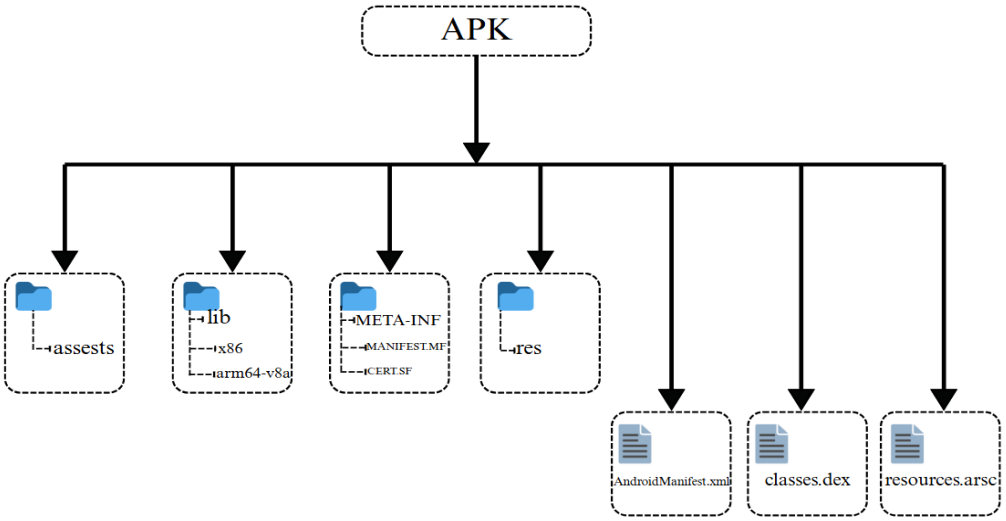

Figure 2 APK File Structure

3.3. Smali Code Transformation

3.3.1. Eliminating Variable Markings

Registers act as storage spaces for temporary data in smali files. The lines of code in Smali files frequently use several registers . These registers are usually named with variables like v0, v1, p0, p1, and so on, and are linked with specific functions being executed. Nevertheless, this comprehensive data regarding storage variables and their linked functions might not clearly assist in the assessment of a program's harmful or harmless character.

3.3.2. Eliminating Special Characters

Smali files often have extra characters that don't really help in giving input to the detection system. These characters are usually ignored by the tokenizer. From Our observations are that when the tokenizer finds special characters in the input, it usually swaps them with spaces . However, it’s important to note that this method doesn’t always work perfectly, because sometimes it can change the original meaning of the code. To solve this issue, we use two strategies for handling special characters: First, during the text cleaning step, any special characters that connect two words or are at the start or end of a word are replaced with spaces.

3.3.3. Eliminating Single-Character Strings

After getting rid of special characters, we notice a lot of single-letter strings appearing inside the smali files. These one-character strings don't really add any important meaning to the code. So, we move forward and remove all these single-length characters too.

3.3.4. Eliminating Comments and Alerts

Inside smali files, there are many notes and notifications meant to give information to the user. These messages are made to appear in pop-up boxes while the program runs. However, these lines don't really help with malware detection and can safely be ignored or deleted.

3.3.5. Eliminating Redundant Lines

After eliminating specific words, characters, and variables in earlier steps of our process, many lines in the smali files become identical.At this stage,having just one of these identical lines is enough ,so all the other repeated lines can safely be deleted.

3.4. Ensemble Classification

To develop a more dependable malware detection model, an ensemble learning approach utilizing Random Forest and XGBoost was implemented Figure 3 These two have strong performance metrics, are efficient in high-dimensional feature sets, and are capable of preventing over-fitting.

- **Random Forest:** An example of a bagging style ensemble, in which many different decision trees are trained, where predictions are based on a majority vote within the ensemble.
- **XGBoost:** An ensemble boosting model where trees are incrementally constructed, and are constructed to mitigate the errors of the previous models. Figure 4 shows Shap Interpretation

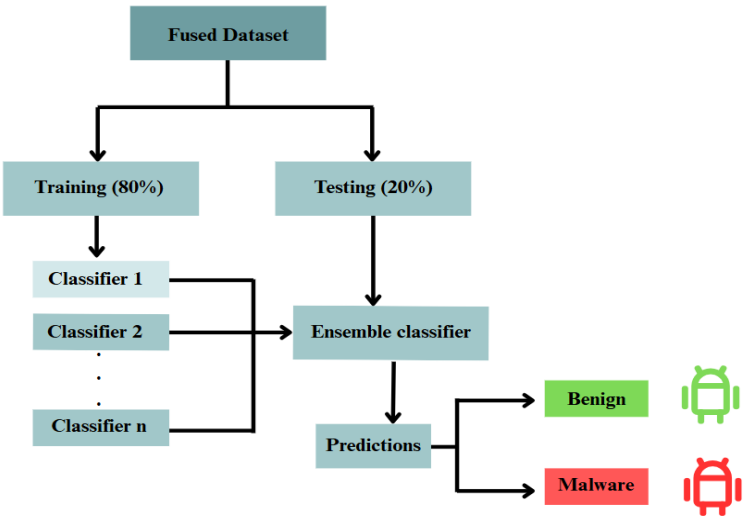

Figure 3 Ensemble Classification

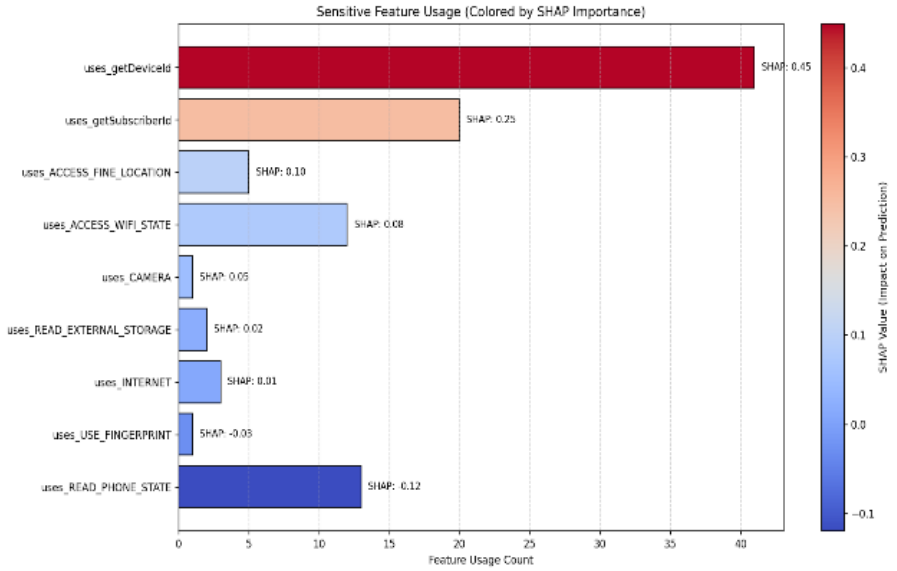

Figure 4 Shap Interpretation

The final method for predicting uses soft voting, whereby both classifiers predict probabilities independently for malware, and the mean of both probabilities is taken to determine the label decision for the APK. Simply put, if the mean probability crosses a pre-specified threshold (e.g., 0.5) than the APK would be labelled malicious, otherwise benign. This ensemble method was significant in minimizing false positives, while increasing detection accuracy. The combined model was tested and returned.

3.5. Interpretation

To improve the interpretability of malware classifications, the SHAP (SHapley Additive exPlanations) framework is incorporated in the last stage of the analysis. SHAP provides an explanation of individual predictions by calculating the contributions of each input feature, concerning the final classification. The color of each bar is encoded with the SHAP value, which reflects how much that feature pushed the model to predict a malicious or benign output in the first place.

- **Positive SHAP (red):** This feature adds to the probability of being malicious.
- **Negative SHAP (blue):** This feature supports benign behavior.

The darker the color, the more influence that feature had on the model's decision.

4. Results and Discussion

4.1. Results

4.1.1. Experimental Setup

The proposed Android malware detection framework was evaluated on a Lenovo laptop equipped with an Intel 5 processor. The implementation was developed in Python 3.8, leveraging open-source libraries such as Androguard, Baksmali, NetworkX, and PyTorch Geometric for Graph Neural Network.

4.1.2. Dataset Description

We used the AndroZoo repository's official API to request access in order to create a custom dataset for malware detection in our investigation. The APK samples were obtained directly through permitted API calls customized to our project's needs, in contrast to pre-collected dumps. Specifically, we focused on collecting APKs from a select few years from 2011 to 2024.For every chosen year, we downloaded 1,000 malicious and 1,000 benign samples. A benign sample was identified by a VirusTotal detection score of 0, whereas a hazardous sample was identified Table 1 shows Ensemble Classification Report

Table 1 Ensemble Classification Report

Label	Precision	recall	F1-score	Support
0	0.999	0.999	0.999	182
1	0.999	0.999	0.999	195
accuracy			0.999	377

by a detection score of 10 or higher. However, network and availability problems prevented some APKs from being downloaded successfully. And number of apks available for particular year .

4.1.3. Evaluation Metrics

The widely used metrics for evaluating the performance of our detection method are Accuracy, Precision, Recall, and F1(F-score) shown in Table 2.

Accuracy = (TP +TN) / (ALL)

Precision = TP/ (TP +FP)

Recall = TP/ (TP +FN)

F1 =2∗Precision∗Recall / (Precision +Recall)

(1) where TP (True Positive) is the number of malicious Android apps that are correctly labeled as malicious, FN (False Negative) is the number of benign Android apps that are falsely labeled as malicious apps. FP (False Positive) is the number of malicious Android apps falsely labeled as benign, and TN (True Negative) is the number of benign Android apps correctly labeled as benign.

4.2. Discussion

The very high classification scores presented in Table 1 show the effectiveness of the proposed ensemble-based detection system. The label 0 indicates benign and 1 indicates malicious. With all of the major performance measures at 0.999, the model has remarkable strength for distinguishing between benign Android applications and malicious ones. This results substantiate the merging of semantic features from Smali code and structural features from Sensitive Function Call Graphs (SFCGs). As displayed in Table 2, ensemble methods in the form of Random Forest and XGBoost performed consistently better in all evaluation metrics we investigated in this study when compared to the other algorithms used. XGBoost yielded a recognition-based F1 score of 99.94%, which in terms of F1 has the best balance between precision and recall. Random Forest followed XGBoost closely with an F1 score of 99.70% and achieved the highest accuracy of algorithms in this study with 99.72%. Therefore, we used Random Forest and XGBoost as the base models in our ensemble framework. Table 2 shows Comparison of Different Models

Table 2 Comparison of Different Models

Models	F1-score	Precision	recall	Accuracy
SVM	96.66	96.74	97.31	95.64
Random Forest	99.70	99.73	99.67	99.72
Decision Tree	98.45	98.56	98.65	98.67
KNN	92.46	92.56	92.36	92.85
XGBoost	99.94	99.15	99.14	99.66
Light GBM	86.78	88.43	85.83	87.82

Conclusion

Our proposed framework achieves effective Android malware detection by capitalising on both structural information with Sensitive Function Call Graphs (SFCGs) and semantic information with Word2Vec-embedded Smali code. The fine-tuning of ensemble classifier set-ups such as Random Forest and XGBoost with soft voting achieves classification capabilities and SHAP explanations improved interpretability by explaining which features mattered most for each classifier decision. Despite high accuracy and transparency, there are large opportunities for future improvements. Most importantly, expanding the dataset to have more APKs across years, obfuscation techniques, and also, dynamic behavioral features (like runtime calls or network traces) could help achieve a more real-time or on-device detection system; further resilience could come from adversarial training.

References

[1]. Gong, J., Niu, W., Li, S., Zhang, M., & Zhang, X. (2024). Sensitive Behavioral Chain-focused Android Malware Detection Fused with AST Semantics. IEEE Transactions on Information Forensics and Security. https:// ieeexplore. ieee.org/ abstract/ document/10695137/

[2]. Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E. D., Ross, G., & Stringhini, G. (2019). MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models (Extended Version). ACM Transactions on Privacy and Security, 22(2), 1–34.

https://doi.org/10.1145/3313391

[3]. Huang, Y., Liu, J., Xiang, X., Wen, P., Wen, S., Chen, Y., Chen, L., & Zhang, Y. (2024). Malware Identification Method in Industrial Control Systems Based on Opcode2vec and CVAE-GAN. Sensors, 24(17), 5518.

[4]. Anand, A., Singh, J. P., & Singh, A. K. (2025). Smali code-based deep learning model for Android malware detection. The Journal of Supercomputing, 81(4), 1–31.

[5]. Cui, L., Cui, J., Ji, Y., Hao, Z., Li, L., & Ding, Z. (2023). API2Vec: Learning Representations of API Sequences for Malware Detection. Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 261–273. https:// doi.org/ 10.1145/3597926.3598054

[6]. Nethala, S., Chopra, P., Kamaluddin, K., Alam, S., Alharbi, S., & Alsaffar, M. (2025). A deep learning-based ensemble framework for robust android malware detection. IEEE Access, 13, 46673–46696.

[7]. Sharma, S., Ahlawat, P., & Khanna, K. (2024). DeepMDFC: A deep learning based android malware detection and family classification method. SECURITY AND PRIVACY, 7(2), e347. https:// doi.org/10.1002/spy2.347

[8]. Faghihi, F., Zulkernine, M., & Ding, S. (2022). CamoDroid: An Android application analysis environment resilient against sandbox evasion. Journal of Systems Architecture, 125, 102452.

[9]. Huang, H., Huang, W., Zhou, Y., Luo, W., & Wang, Y. (2025). FEdroid: A lightweight and interpretable machine learning-based android malware detection system. Cluster Computing, 28(4), 224.

[10]. Gu, J., Zhu, H., Han, Z., Li, X., & Zhao, J. (2024). GSEDroid: GNN-based android malware detection framework using lightweight semantic embedding. Computers & Security, 140, 103807.

[11]. Nasser, A. R., Hasan, A. M., & Humaidi, A. J. (2024). DL-AMDet: Deep learning-based malware detector for android. Intelligent Systems with Applications, 21, 200318.

[12]. Alzaylaee, M. K., Yerima, S. Y., & Sezer, S. (2020). DL-Droid: Deep learning based android malware detection using real devices. Computers & Security, 89, 101663.

[13]. Ma, Z., Ge, H., Wang, Z., Liu, Y., & Liu, X. (2020). Droidetec: Android Malware Detection and Malicious Code Localization through Deep Learning (No. arXiv:2002.03594). arXiv. https:// doi.org/ 10.48550/arXiv.2002.03594

[14]. Liu, X., Liu, X., Hao, K., Wang, K., Chen, X., & Niu, W. (2024). HGNNDroid: Android Malware Detection Based on Heterogeneous Graph Neural Network. 2024 IEEE 9th International Conference on Data Science in Cyberspace (DSC), 378–384. https:// ieeexplore. ieee.org/ abstract/ document/10859039/

[15]. Wu, Y., Li, X., Zou, D., Yang, W., Zhang, X., & Jin, H. (2019). Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis. 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 139–150. https:// ieeexplore. ieee.org/ abstract/document/8952382/