Static Analysis-based Android Malware Detection (2021–2026): A Systematic Survey and Taxonomy / Review Article

Assist. Lact. Ali Hussein Mohammed Ali[1*]

and Assist. Lact. Musaab Riyadh Abdulrazzaq[2]

1,2 Department of Computer Science,
College of Science, Mustansiriyah University, Baghdad/Iraq.

*** Corresponding Author: abo.shbeeb90@gmail.com**

الكشف عن البرامج الضارة في نظام أندرويد باستخدام التحليل الثابت (2021-2026): دراسة منهجية وتصنيف

م. م. علي حسين محمد علي[1*] و م.م. مصعب رياض عبدالرزاق [2]

1, 2 الجامعة المستنصرية، كلية العلوم، قسم علوم الحاسوب، بغداد|العراق.

Abstract

Android malware is not only increasing in size and sophistication but is also a challenge to be detected on a large scale. Static analysis is popular due to its ability to detect malware without running applications or tracing run-time events. However, features, datasets, labeling methods, and assessment methodology differ, which makes studying performance difficult. This is a systematic study of 78 peer reviewed articles written between 2021 and 2026 regarding the Android malware detection by static analysis. In each of the studies, the current survey cover the following feature source, feature representation and engineering, learning paradigms, datasets and labeling methods, and evaluation configuration Permissions, API-based features, opcode-bytecode indicators, manifest-component indicators, metadata-certificate indicators, hybrid static features, adversarial and robustness-oriented designs, graph-based designs, and transformer-based models are represented. Synthesis trade-offs include larger semantic and structural representation, validity risks, that lack of obfuscation resistance is not evaluated, and reporting of repeatability is poor. This study suggests an evidence-based taxonomy and comparative synthesis of static detection methods, structured dataset landscape for decision making, and gap analysis for time sensitive evaluation, leakage prevention, robust benchmarking and repeatable reporting. These findings point to building and experimenting with static detectors for Android malware.

Keywords: Android malware detection, Static analysis permissions, API calls, Opcode, Graph neural networks.

المستخلص

لا يقتصر تزايد حجم وتعقيد البرمجيات الخبيثة لنظام أندرويد على ذلك فحسب، بل يُمثل أيضًا تحديًا كبيرًا لاكتشافها على نطاق واسع. يُعد التحليل الثابت شائعًا لقدرته على اكتشاف البرمجيات الخبيثة دون تشغيل التطبيقات أو تتبع أحداث وقت التشغيل. ومع ذلك، تختلف الميزات ومجموعات البيانات وأساليب التصنيف ومنهجية التقييم، مما يُصعّب دراسة الأداء. هذه دراسة منهجية لـ 78 مقالة مُحكّمة نُشرت بين عامي 2021 و2026 حول اكتشاف البرمجيات الخبيثة لنظام أندرويد باستخدام التحليل الثابت. في كل دراسة، يُغطي هذا المسح الحالي مصادر الميزات، وتمثيلها وهندستها، ونماذج التعلم، ومجموعات البيانات وأساليب التصنيف، وتكوين التقييم. كما يُمثل الأذونات، والميزات القائمة على واجهة برمجة التطبيقات، ومؤشرات رمز العملية/رمز البايت، ومؤشرات البيان/المكون، ومؤشرات البيانات الوصفية/الشهادة، والميزات الثابتة الهجينة، والتصاميم المُوجّهة نحو الخصوم والمتانة، والتصاميم القائمة على الرسوم البيانية، والنماذج القائمة على المحولات. تشمل المفاضلات في عملية التركيب تمثيلاً دلالياً وبنيوياً أوسع، ومخاطر تتعلق بصحة النتائج، وعدم تقييم مقاومة التمويه، وضعف الإبلاغ عن قابلية التكرار. تقترح هذه الدراسة تصنيفاً قائماً على الأدلة وتركيباً مقارناً لأساليب الكشف الثابتة، وهيكلة بيانات منظمة لدعم اتخاذ القرارات، وتحليلاً للفجوات من أجل التقييم الحساس للوقت، ومنع التسريب، ووضع معايير قياس قوية، وإعداد تقارير قابلة للتكرار. تشير هذه النتائج إلى أهمية بناء أدوات كشف ثابتة للبرمجيات الخبيثة لنظام أندرويد وتجربتها.

الكلمات المفتاحية: كشف البرامج الضارة للاندرويد، أذونات التحليل الثابت، مكالمات APK ، رمز التشغيل، الرسم البياني للشبكات العصبية.

1. Introduction

Android is the operating system that has acquired a dominant position in the mobile ecosystem because of its open-source software and wide use on a device [1]. According to numerous industry reports and user statistics, it remains widespread all over the world, with Android being reported as the most popular mobile OS in 2019 [2]. It is also being adopted in Iraq approximately 81% as compared to the global share 87%, according to [3]. The Android OS usage in Iraq is shown in Figure (1). The Android ecosystem is huge, and its openness as well as broad adoption of third-party libraries and SDKs has increased the attack surface and created new security risks [4]. Practically, users often download software provided by various markets and vendors, which exposes them to repackaged or malicious software and early detection of legitimate and malicious software is a crucial feature of securing user privacy and integrity of their devices [5]. Signing and certificate tools at the application layer give valuable security assurances, but do not entirely exist capable of thwarting malware spread, particularly in the face of fast malware development and supply-chain-like channels of distribution [6]. Malware has also developed since the early days of the mobile attacks to massive campaigns with high expansion in Android malware in the last ten years [7].

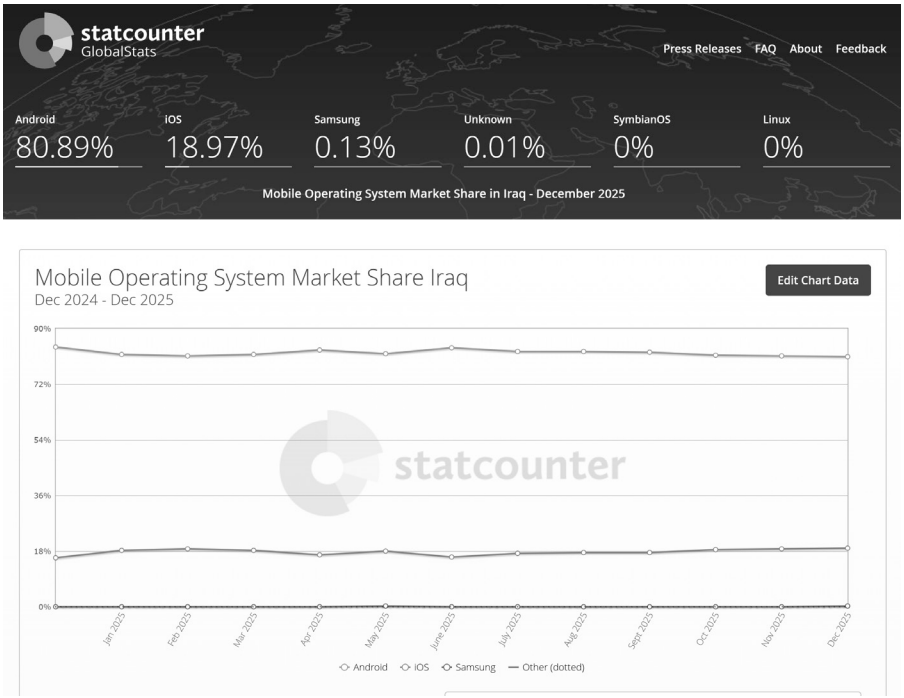

Figure (1): Android Popularity in Iraq

On-device security layers and app stores have broad usage of detection methods to label applications as either benign or malicious and reduce the risks of malware before they are downloaded. Overall, there are malware detection methods that are divided into the categories of static, dynamic, and hybrid malware detection strategies [4]. Static detection is used to analyze application artifacts without running the application, dynamic detection is used to examine the behavior of an application during execution, and hybrid detection is used to bring together the two views [8]. Simultaneously, malware can also be analyzed at the family level, as families are collections of samples with similar behavioral and structural characteristics, family-level knowledge is useful in prioritizing and analyzing samples beyond isolated samples [6]. Android malware detection static analysis is a feasible and popular methodology

due to its ability to conduct the assessment with high speed, even in case they are not run, which is advantageous in the context of large-scale screening and analysis without leakage of privacy information [9]. The current review takes into consideration the use of the static detection of the various application artifacts, such as permissions [9], API usage [10], manifest components, and certificate/metadata cues, which are useful in efficient triage and model training. However, modern evasion techniques like obfuscation, packing/encryption, reflection and repackaging inherently challenge the operation of static detectors in terms of loss of feature fidelity and undermining generalization across time and markets. These properties drive a systematic synthesis of feature selections, representations, and modeling paradigms throughout recent literature on the topic of static analysis. The contributions of this review are as follows:

- It systematically surveys 78 peer-reviewed studies (2021–2026) on Android malware detection using static analysis only, and consolidates their feature sources, representations, models, datasets, and evaluation protocols into a unified evidence base.
- It proposes an evidence-driven taxonomy that organizes static detection research across permissions, APIs, opcodes/bytecode, manifest/components, metadata/certificates, hybrid static features, robustness-oriented designs, graph-based methods, and transformer-based models.
- It provides a comparative synthesis that highlights cross-study trends and trade-offs (accuracy vs. scalability, semantic richness vs. engineering cost, robustness vs. obfuscation).
- It identifies methodological gaps and threats to validity (dataset aging, label noise, inconsistent evaluation setups, and limited reproducibility reporting) and derives a forward-looking research agenda for more reliable static detection.

Some recent surveys have surveyed Android malware detection, although most related literature either (i) is biased to the broad machine-learning view of

mixed feature families and at times mixed sources of knowledges (static and dynamic), (ii) is concerned with adjacent tasks, such as malware family classification, identification of third-party libraries, or testing of Android applications, or (iii) is out of date with the high rate of posteri-facto evolution of the static-analysis approach in recent years. By contrast, this paper gives a recent, static analysis-only synthesis of the 2021-2025 period and classifies the literature using an evidence-based taxonomy making modern directions of statical analysis first-class categories (graph/structural representations, robustness/adversarial considerations, and transformer-based static models). In addition to categorization, this study groups the dataset landscape into a decision-oriented reference and provides a critical examination of the threat to the fair comparable cross study, label reliability (variation in antivirus consensus), evaluation protocol variations, and report voids, to make the readers of the reported accuracy claims more realistic. Table I summarizes these differences and explains the added value of this survey as compared to similar reviews.

Table I: Summary of compared surveys

Study		This study	[2]	[4]	[5]	[6]	[7]	[8]	[1]
Checkpoint	**Year**	**2025**	**2020**	**2020**	**2024**	**2020**	**2024**	**2021**	**2025**
Android-focused		✓	✓	✓	✓	✓	✓	✓	✗
Android malware detection focus		✓	✓	✓	✓	✓	✗	✗	✓
Static-analysis-only scope		✓	✗	✗	✗	✗	✗	✗	✗
Coverage window: 2021–2025		✓	✗	✗	✗	✗	✗	✗	✓
Explicit SLR workflow		✓	✗	✗	✓	✗	✓	✗	✓
Fine-grained static taxonomy		✓	✗	✗	✗	✗	✗	✗	✗
Hybrid static feature fusion category		✓	✗	✗	✗	✗	✗	✗	✗
Graph/structural static category		✓	✗	✗	✗	✗	✗	✗	✗
Adversarial/robustness category		✓	✗	✗	✗	✗	✗	✗	✗
Transformer-based static category		✓	✗	✗	✗	✗	✗	✗	✗
Decision-grade dataset landscape		✓	✗	✗	✓	✗	✓	✗	✗
Comparability/label reliability critique		✓	✗	✗	✗	✗	✗	✗	✗

To support navigation, the remaining parts of this paper are structured as follows: Section 2 explains the systematic review protocol. The methodology-driven taxonomy and comparative synthesis of static detection categories is presented in section 3. Section 4 is an analysis of the datasets used in reviewed studies. Section 5 focuses on cross cutting gaps and threats to validity. Finally, section 6 summarizes evidence-driven recommendations and a research roadmap for future research.

2. Systematic Protocol of Literature

This research is conducted after the systematic literature review (SLR) protocol with a screening workflow to find the studies on Android malware detection using static analysis. Five major digital libraries were searched because of their broad coverage of computer science, engineering, and security research literature: ScienceDirect (Elsevier), IEEE Xplore, Web of Science (WOS), SpringerLink, and Scopus.

2.1 Search Strategy

The following query was used with minor syntax adaptations to match each database's advanced search format:

("Android") AND ("Malware" OR "Malicious Software") AND ("Static Analysis" OR "Static" OR "Static analysis") AND ("Classification" OR "Detection"). The search was conducted in November 2025 across the selected databases and restricted to English-language articles published between 2021 and 2025. The review is specifically narrowed down to peer-reviewed journal articles to achieve methodological rigor, full reporting of the experiment, and archival stability of the evidence of 2021-2025. Journal research is generally more comprehensive in its descriptions of datasets, labeling methods, assessment processes and details of

reproducibility, so it can be more dependable on cross-study synthesis and lessen noise due to early or informal reporting. Consequently, the results of this SLR are expected to be used by researchers and practitioners as a high-confidence source of information about deployment-relevant findings of well-established, proven methods of static-analysis. The retrieved records from all databases were consolidated and processed through consecutive screening and eligibility steps, as summarized in Figure (2).

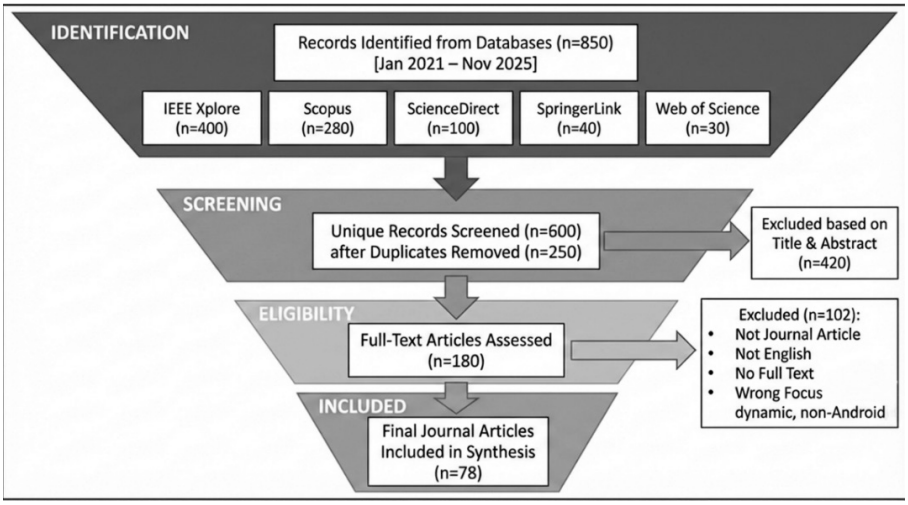

Figure (2): Research query results

2.2 Study Selection Process

A total of 850 records were identified across the five databases (ScienceDirect: 100, Scopus: 280, IEEE Xplore: 400, WOS: 30, SpringerLink: 40). After removing duplicates, 600 unique records remained. Title and abstract screening reduced the set to 420 potentially relevant studies. Full-text assessment and the application of inclusion/exclusion criteria yielded 180 eligible studies, from which 78 articles were retained as the final set for synthesis.

2.3 Inclusion Criteria

Studies were included if they met all the following conditions:

1. Published in a peer-reviewed journal.
2. Focused on Android malware detection using static analysis techniques (permissions, API calls, opcodes/bytecode, manifest/components, certificates/metadata, graphs, or hybrid static features).
3. Available as full text online.
4. Written in English.
5. Published within 2021–2025 to reflect contemporary detection pipelines, datasets, and ML/DL architectures.

2.4 Exclusion Criteria

Studies were excluded if any of the following applied:

1. Not written in English.
2. Not accessible as full text through the selected databases.
3. Focused on malware for other operating systems like (Windows, Linux, MAC) rather than Android.
4. Primarily relied on dynamic analysis or runtime traces.

2.5 Research Questions

This survey aims to answer the following research questions within static-analysis-based Android malware detection:

- **RQ1**: What types of static detection techniques are used across the selected studies?
- **RQ2**: What are the strengths and limitations associated with each static technique?

- **RQ3**: What algorithms and learning models are employed (ML/DL/ hybrid), and how are features represented?
- **RQ4**: What datasets are used (public/private/ benign/malware sources), and what are the sample sizes and labeling strategies?
- **RQ5**: What are the key motivations, contributions, and reported performance outcomes of each study?

2.6 Quality Assessment (QA) Criteria

The QA scores were used to interpret findings and to contextualize unusually high-performance claims, rather than excluding studies solely based on score. To ensure that the synthesis reflects methodologically reliable evidence, this review applied a lightweight quality assessment rubric. Each included study was scored across the following criteria using a 3-point scale (0 = not addressed, 1 = partially addressed, 2 = clearly addressed):

- **QA1**: Clear problem definition and threat model (malware type, detection objective).
- **QA2**: Static-analysis pipeline clarity (feature source, extraction steps, representation).
- **QA3**: Dataset transparency (source, size, class balance, de-duplication).
- **QA4**: Labeling reliability (label source, VirusTotal/AV thresholds if used, scan date if reported).
- **QA5**: Evaluation rigor (split strategy, validation protocol, leakage controls, time-aware evaluation if applicable).
- **QA6**: Reproducibility signals (tool availability, parameter reporting, code/dataset accessibility).
- **QA7**: Robustness considerations (obfuscation, reflection/dynamic loading discussion, limitations).

2.7 Data Extraction and Synthesis Plan

For each included study, this review extracted a standardized set of fields to support evidence-driven comparison and synthesis:

- **Methodology**: feature source(s) (permissions, APIs, opcodes/bytecode, manifest/components, metadata/certificates, graphs), representation (vector/sequence/graph/embedding), and pipeline/tools (static analysis framework, decompilation tools if stated).
- **Modeling**: learning paradigm (ML/DL/GNN/Transformer), classifier architecture, feature selection/engineering steps.
- **Datasets**: dataset names, sample sizes, class balance, benign/malware sources, family vs binary labels, duplication, and timeframe when reported.
- **Evaluation**: split strategy (random/time-based/family-disjoint), metrics (accuracy, precision, recall, F1, AUC), and baselines.
- **Limitations and validity**: obfuscation handling, labeling uncertainty, drift concerns, leakage risks, and reproducibility constraints.

3. Methodology and Taxonomy Analysis

This systematic review synthesizes 78 peer-reviewed studies published between 2021 and 2025 that address Android malware detection using static analysis only without executing applications or collecting runtime traces. To support evidence-driven synthesis, each study was coded using a consistent extraction form covering (i) the primary static signal source, (ii) the adopted representation or feature engineering, (iii) the modeling paradigm, (iv) the dataset and labeling strategy, (v) the evaluation protocol, and (vi) the reported limitations

The taxonomy categories are not necessarily mutually exclusive, since in practice, many studies use a combination of multiple immobile signals such as permissions and API calls. Each study is coded with one primary category depending on the most dominant or most distinguishing signal and representation, and other ones are listed as secondary tags to prevent the scenario of counting the same study under different categories. Thus, the total counts of categories and the per-category evidence tables are based on primary-category assignment only, and hybrid multi-signal studies are recorded once under the Hybrid category and are not repeated under their respective categories. Accordingly, the total category counts and the per-category evidence tables will indicate the primary-category assignment only, and hybrid multi-signal studies will be counted once under the Hybrid category and will not be counted again under all their respective categories. As summarized in Figure (3), the reviewed literature is first organized by the dominant signal source into: (a) manifest- and metadata-level signals (N=18), (b) code-level signals (N=23), and (c) hybrid and advanced paradigms (N=37) that integrate multiple feature sources and emphasize modern modeling and evaluation dimensions. Within each group, the subsequent subsections provide evidence tables mapping individual studies to their representations, model families, enabling explicit comparisons of design trade-offs such as interpretability versus feature complexity, accuracy versus scalability, and robustness under obfuscation. Across the surveyed studies, recurring limitations include sensitivity to code obfuscation and reflection/dynamic loading, reliance on non-time-aware benchmarks, and incomplete reporting that limits reproducibility.

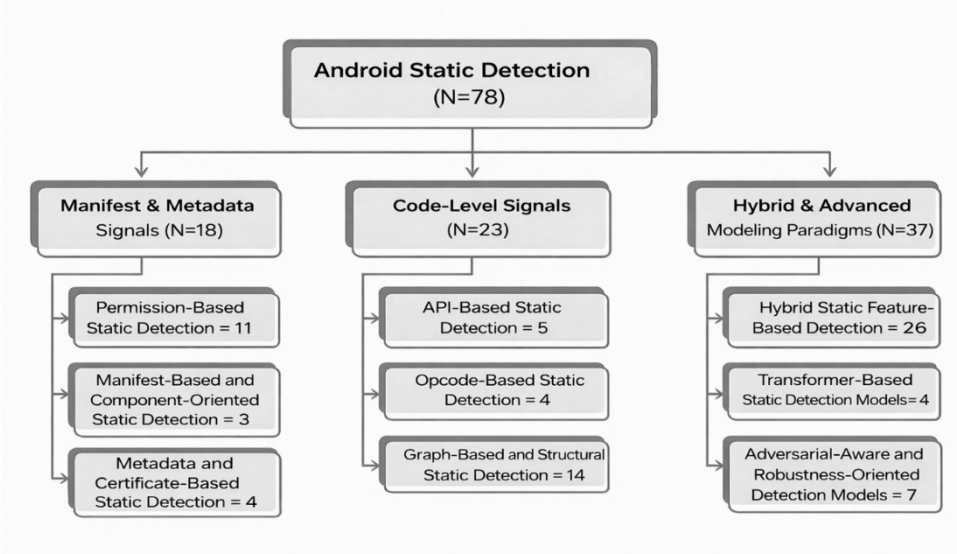

Figure (3): Taxonomy Analysis

3.1 Permission-Based Static Detection

Permission-based static detection is a core component of Android security research, which is concerned with the analysis of application requested privileges in the AndroidManifest.xml file or their use in the code. Several studies have been done on how to optimize standard machine learning classifiers by good feature selection. A model proposed in [9] focuses on avoiding reliance on dynamic execution and extracting permission features from the manifest which can achieve 91.1% accuracy using a combination of feature selection and Bayesian classification. Validating the validity of the standard classifiers, the authors in [11] used a balanced data set to show that Random Forest and SVM classifiers could achieve roughly 93.75% accuracy using binary permission vectors, although the authors acknowledged differences in terms of obfuscation. To overcome the issue of data freshness and feature redundancy, the framework proposed in [12] used feature

importance guided attribute reduction to reduce the feature space from 323 to 48 key permissions so that a Random Forest classifier achieved 97.5% accuracy. Moving past the question of whether permissions exist or not, researchers have explored the structural relationships between permissions. The approach in [13] proposed the Composition Ratio (CR) where the ratio of permission pair occurrences to the total number of pairs is calculated to form the feature vector that can be robust to permission inflation and noise. These structural approaches show that permission co-usage modeling gives a better discriminative power than viewing permissions as independent binary features [13].

Recognizing the fact that the legitimacy of permission request often depends on the function of an application, recent works have integrated category awareness in detection models. A study utilizing Self-Organizing Map (SOM) and K-means clustering mined the permission usage patterns specific to the app categories and used these patterns to identify malware that requests permissions incoherent with the declared category [14]. Similarly, the authors in [15] have considered a permission sensitivity-based detector that assigns weights to permissions according to their overload or lost status into a particular category and achieves 92.17% accuracy by differentiating permissions to benign and malicious apps in context. Furthermore, the framework in [16] uses Collaborative Filtering to determine unnecessary permissions based on the category of an application, and they neutralize them by injecting code to return synthetic data to prevent privacy leakage while keeping the application from crashing. In addition, the study [17] was focused on Real Permissions which were extracted using reverse engineering and the FP-Growth algorithm to find correlated pairs of permissions invoked at runtime, which were used to train a Neural Network. Special architecture has been created to overcome efficiency limitations and complex threat situations presented in [18].

LinRegDroid has shown that the permission-based detection could be successfully addressed by simple and efficient multiple linear regression models with performance like complex classifiers such as SVM and Decision Trees, but computationally feasible for on-device deployment [18]. Addressing distributed attacks, the SigColDroid was built to identify colluding applications, and it uses dangerous permissions in conjunction with permission rates and Smali file size, to successfully identify colluding malware from lone apps with a high ROC_AUC of 99.48% [19]. Finally, authors in [20] make a baseline comparison of ML models on permission features using 331 permission indicators with several Machine Learning algorithms. In order to get final thought of this approach, Table II represents contributions of each article related to this approach

Table II: Permission-Based Article Contributions

Study ID	Year	Permission signal	Representation / feature engineering	Classifier / decision model
[9]	2022	Manifest permissions	Permission subset selection (feature selection)	Bayesian classification
[13]	2021	Manifest permissions	Permission-pair Composition Ratio (CR) vector (ratio-based)	Not specified
[17]	2023	Real permissions used in code	Reverse engineering → real permissions, FP-Growth to mine correlated permission pairs, semantic couples	Neural Network
[14]	2022	Manifest permissions + category context	103-dim binary + SOM + K-means to mine category-specific permission patterns, pattern-fit features	SVM
[18]	2022	Manifest permissions	Binary vectors, multiple linear regression as risk score, 2 rule-based classifiers + bagging + hybrid ensemble	Regression-based + ensembles (incl. SVM/DT in ensemble-2)

Study ID	Year	Permission signal	Representation / feature engineering	Classifier / decision model
[15]	2021	Manifest permissions + category context	Category inference from description (NB/TextCNN), Permission Sensitivity (PS) weighting (overload/lost permissions)	RF, SVM, k-NN
[16]	2022	Manifest permissions	Collaborative Filtering + Frequent Permission Set Mining to find unnecessary permissions, instrumentation to return fake data	Not a pure classifier (permission minimization + runtime hooking)
[11]	2025	Manifest permissions	Binary permission vectors	RF, SVM, DT
[12]	2024	Manifest permissions	323-permission inventory → feature-importance reduction to 48 permissions	RF, KNN, NB, DT
[19]	2025	Dangerous permissions + static intensity signals	Dangerous permissions + Smali file size + permission rates	RF, ExtraTrees, AdaBoost, XGBoost, LightGBM, ANN, DNN (multi-class)
[20]	2024	Permissions	331 permission indicators, GridSearchCV + 5-fold CV	RF/SVM/XGB/DT

Permission-based detectors often appear effective in laboratory evaluations because manifest permissions are easy to extract and frequently align with malware labels in popular benchmarks; however, this alignment does not necessarily constitute a reliable behavioral signature. Most studies in this evidence base rely on manifest-only permission vectors (7/11), while smaller subsets incorporate category context (2/11), code-derived usage-aware permissions (1/11), or longer intensity-style signals (1/11). This suggests that many reported improvements depend on a proxy that is sensitive to dataset choice and may drift as Android and the app ecosystem evolve. Moreover, the Android permission model has evolved substantially; since Android 6.0, dangerous permissions are granted at runtime,

which weakens the assumption that manifest-declared permissions always translate into consistent capabilities across versions and years. Accordingly, time-sensitive evaluation is increasingly important, as time-based splits better reflect concept drift and platform transitions than conventional random splits. Random splits can also inflate apparent generalization, especially when near-duplicate apps or minor variants leak across training and test folds. This category is not strictly isolated: permission features frequently co-occur with manifest and component indicators and sometimes with code and API cues, which complicates attributing performance gains to permission alone. Overall, the evidence points to a clear trade-off between interpretability and behavioral specificity: manifest-only permission vectors are intuitive and inexpensive, yet they remain coarse proxies that can flag legitimate high-privilege apps and miss stealthier malware that requests fewer permissions. The literature's inconsistencies are largely explained by differences in representation binary vs co-usage vs context-aware weighting and evaluation, with random splits often yielding optimistic estimates compared to time-aware splits that better reflect drift and platform changes.

3.2 API-Based Static Detection

API-based static detection is centered on analyzing the Application Programming Interface (API) calls that are made by an application to determine malicious behavior. While initial methods have focused on naive frequency counting, recent studies have made progress in sequence modeling, semantic clustering, and hybrid deep learning models to understand the contextual use of APIs. Several studies have gone beyond considering APIs as an unordered group of words but instead use deep learning to model the temporal and sequential nature of code execution. One of them uses API call sequences as a time series sequence data

using a Skip-gram model to learn dense numeric embeddings and a Bidirectional Long Short-Term Memory (Bi-LSTM) network to capture long-range dependencies and dependency patterns in the forward and backward directions [10]. Expanding the feature scope Another study suggested a multi-input deep learning model that combines API calls with permissions, opcodes, and fuzzy hashes into a multi-branch Keras Functional API network, achieving around 99.5% accuracy by learning across various static feature types all at once [21]. To overcome the semantic gap and the evolution of the Android platform, researchers have devised methods that pay attention to the meaning and association of the API calls, and not only to their raw occurrence. The SDAC framework solves this problem of model aging and concept drift by embedding APIs in a vector space and grouping them in semantic clusters using k-means, which enables the system to map new or renamed APIs to existing clusters without redesigning the feature space [22]. Recent study also emphasizes optimizing classifier performance and ensuring robustness against common evasion techniques like obfuscation. In addition, to combat API obfuscation and APK size bias, a hybrid framework utilizing DeGuard was proposed in [23] to reveal concealed API calls. It combines these deobfuscated features with certificate metadata and interaction-based ratios to eliminate size bias, significantly outperforming baselines on the Drebin dataset. Finaly, to reduce false alarms and improve precision, an automated framework was introduced in [24]. It utilizes metaheuristic algorithms such as Binary Coati Optimization to select discriminative API features and tune the hyperparameters of a stacking ensemble classifier and achieving over 98% accuracy. In order to get final thought of this approach, Table III represents contributions of each article related to this approach.

Table III: Article contributions in API-based

Study ID	Year	API-based static signal	Representation / feature engineering	Model family	Other static signals used	Task (label space)
[21]	2022	API calls (among multi-signals)	Multi-branch feature inputs fused before softmax (Keras Functional API)	Multi-input deep learning	Permissions, services, receivers, opcodes, file size, fuzzy hashes (ssdeep)	Binary + 5-class (Benign/SMS/ Banking/Adware/ Riskware)
[24]	2023	API calls + permissions	Feature subset selection and/or ensemble tuning via metaheuristic (Binary Coati Optimization)	Optimized ensemble (stacking/voting)	Permissions	Malware detection
[23]	2021	API calls (original + deobfuscated)	DeGuard to recover ProGuard-obfuscated APIs, interaction-ratio features to reduce APK size bias	Supervised ML (LightGBM/ CatBoost/RF/ET/ Linear SVM)	Permissions, components (services/activities/intents), certificate features, Apktool log metadata	Malware detection
[10]	2024	API call sequences (path-based)	Call-graph + DFS path extraction → ordered API sequences → n-grams → Skip-gram (Word2Vec) embeddings	Deep sequence model (Bi-LSTM)	N/A	Malware detection
[22]	2022	API semantics under evolution	API path extraction + Skip-gram embeddings → k-means semantic API clusters, unseen APIs mapped to clusters (FEO) or clusters+models updated (FMU)	Ensemble/voting over classical classifiers	N/A	Malware detection under concept drift

In API-based research, the key trade-off is between semantic richness and fragility of extraction: those representations, in which one can maintain order and meaning, are able to enhance specificity of behavior, though at the price of higher computational cost. The literature in this evidence set (n = 5) narrows down to 3/5 multi-signal hybrids, 1/5 sequence-aware modeling, 1/5 semantic/drift-aware abstraction with each having a different trade-off between fidelity and practice. Resilience Hybrid pipelines can provide good laboratory accuracy by combining APIs with opposing static signals but aggregated or set-based API encodings are likely to lose behavioral context and be confounded by third-party libraries and benign applications with large functional footprint. Sequence-sensitive techniques maintain execution-based order and have the ability to represent higher-order dependencies, but are limited by call-path recovery, reflection/dynamic loading, and incomplete static resolution. Semantic drift-aware abstraction alleviates sparsity and API history by relocating new/renamed APIs to consistent clusters, whereas it can obscure malicious intent at a fine-grain when clusters become too large or because labeling evaluation environments change. The discrepant findings between studies are mostly due to differences in library filtering and deobfuscation assumptions, and evaluation conditions which extend to random splits rather than environments which are more akin to drift and deployment constraints. Thus, to have fair synthesis, there must be transparent reporting of the selected API representation, library deobfuscation treatment, and leakage-resistant evaluation, to ensure that the gains are due to the method, and not to artifacts of data sets.

3.3 Opcode-Based Static Detection

Opcode and bytecode analysis beyond metadata or API calls by taking a closer look at the low-level instructions that are run by Android Runtime (Dalvik or ART). Research in this domain has spread into a variety of contexts such as deep sequence

modeling, new signal transformations, and formal semantic verification. In order to solve the shortcomings of the traditional sequence analysis method, researchers have used advanced deep learning architectures to process opcode and bytecode stream. A prominent trend in literature in recent years is the conversion of bytecode into other forms of data in order to exploit algorithms from other areas. A few studies have used bytecode as image data. Similarly, Dex bytecode into RGB image matrices and utilizing self-supervised learning (DINO) with a teacher-student distillation process, which achieves great generalization capability for discovering new families with a limited amount of labeled data [25]. Exploring signal processing, another framework restates malware analysis as an audio classification problem by transforming raw APK bytes into .wav file and extracting engineering features such as MFCCs and spectral descriptors [26]. Moving away from platform-specific parsing, the DAEMON classifier utilizes raw byte N-grams to create a platform-independent detection mechanism [27]. By employing an entropy-based filter to remove noisy data and an Aho–Corasick algorithm for fast mapping, DAEMON demonstrates that a unified, byte-level feature extraction pipeline can achieve accuracy (98.74%) on Drebin Dataset and provide explainability without parsing specific Android file structures like classes.dex [27]. In contrast to statistical machine learning approaches, some research relies on formal verification techniques. A formal equivalence checking framework was proposed that translate Java bytecode instructions into CCS processes and labeled transition systems performing malware analysis by applying weak bisimulation equivalence to compare automata states [28]. This method identifies malware families based on opcode-level behavioral semantics rather than signature matching, making it inherently robust to code obfuscation and syntactic changes without requiring labeled training data [28]. In order to get final thought of this approach, Table IV represents contributions of each article related to this approach.

Table IV: Article contributions in opcode-based

Study ID	Year	Opcode/ byte-level signal	Representation / feature engineering	Model family	Platform scope	Task
[26]	2023	Raw APK bytes (reframed)	APK bytes → .wav audio, extract MFCCs + spectral features, FS via CFS/ReliefF/IG/ Gain Ratio	Classical ML (RF/ SVM/KNN/ J48/ NB)	Android	Malware family classification
[27]	2021	Raw byte N-grams (4/8/16/32)	Entropy filter + multi-stage feature mining, pairwise separation + tags, fast mapping via Aho–Corasick	Random Forest	Cross-platform (Windows PE + Android APK)	Malware classification (and some family-separation logic via tags)
[25]	2025	classes.dex bytecode → image-like signal	Bytecode hex stream → RGB image matrix, self-supervised DINO pretraining, fine-tune with rotation loss + Manifold Mixup, prototypical (metric) few-shot with multi-crop aggregation	Few-shot + self-supervised representation learning	Android	Malware family recognition under few-shot
[28]	2021	Java bytecode semantics (opcode-level behavior)	Bytecode → CCS processes → labeled transition systems, weak bisimulation for equivalence, heuristic pruning reduces comparisons	Formal verification (no supervised ML)	Android	Malware detection + family classification via equivalence

In studies based on opcodes-bytecodes, a major trade-off has been one of robustness versus interpretability: low-level instructional signals are less sensitive to superficial code change but give less insight into the semantic explanation of why an app is malicious. In this evidence set (n = 4), there are four paradigms of representation: raw byte n-grams (1/4), signal reframing (1/4), bytecode-to-image self-supervised representation learning (1/4) and opcode-level formal semantics through equivalence reasoning (1/4). Three of the four papers shift the focus on reframing, self-supervised representation learning, or semantic equivalence reasoning, directly in an attempt to be less sensitive to superficial code changes and common semantic obfuscations. But working at the level of the byte/opcode level can lose interpretability and it can obscure benign complexity or third-party library space with malicious intent resulting in errors that are more difficult to diagnose. Furthermore, cross-study-comparison is limited by the inability to match tasks like binary vs family vs few-shot, that is, near-ceiling results can indicate an easier or a different task, not a universal high-quality representation. Thus, comparability needs to have equal task set-ups and record the labeling and assessment set up, as well as the particular transformation and representation pipeline that determines robustness in each study.

3.4 Manifest-Based and Component-Oriented Static Detection

Manifest-based and component-oriented detection strategies do analyze the structural backbone of the applications on Android. By looking at declared components and interactions between them, these methods attempt to detect the presence of malicious intent by detecting anomalies in the configuration, modeling the capability, and tracking inter-app communication patterns. The framework presented in [29] uses an extended π-calculus to model Android components as

processes and their interactions as actions and defines precise operational semantics for the evolution of components. By establishing ideas of behavioural equivalence between an application's actual behaviour and templates of known malicious behaviour, this approach offers a solid decision-making framework for identifying complex threats such as collusion, even without machine learning [29]. Instead of using two classes (benign vs. malicious), recent research focuses on what an application can do. These focuses improve dealing with the high dimensionality of manifest attributes, and optimization techniques are often used. BFEDroid [30] is dedicated to feature selection across the manifest-level attributes and ranks the permissions and the intent filters to eliminate noise. By training traditional classifiers on these reduced subsets, the framework shows that a smaller optimized feature space can maintain and improve the detection accuracy while greatly decreasing the model complexity and training time [30]. Furthermore, detecting collusion between applications requires analyzing how components communicate across app boundaries and integrating diverse static features often yields the most robust results. SEDMDroid in [31] proposes a stacking ensemble detector that fuses permissions, sensitive APIs, manifest events, and taint-style data flows into a unified framework. By training Multi-Layer Perceptron base learners on these heterogeneous features and combining them with an SVM meta-classifier, the system achieves approximately 94.9% accuracy when data-flow features are included, consistently outperforming individual classifiers and single-view models [31]. In order to get final thought of this approach, Table V represents contributions of each article related to this approach.

Table V: Article contributions in Manifest-based

Study ID	Year	Component / manifest-centric signal	Representation / analysis method	Model family	Task
[29]	2022	Android components + ICC semantics (Activities/Services/ Receivers/Providers, intents/data)	Extended π-calculus with operational semantics, behavioral equivalence (strong/weak simulation, bisimulation) against benign/malicious templates, decision rules (credible/malicious/ suspicious)	Formal methods (no ML)	Template-based behavioral classification, collusion reasoning
[30]	2022	Manifest permissions + intent filters	Feature ranking/selection over permissions+intents (BFEDroid) to reduce noisy/redundant manifest attributes	SVM, DT, NB, RF	Malware detection
[31]	2021	Manifest events + permissions + sensitive APIs + permission-rate + taint-style source–sink flows	Decompile + manifest parsing, FlowDroid for flows, high-dimensional vectors, stacking ensemble: multiple MLP base learners + SVM meta-classifier, diversity via subspaces/bootstraps/ PCA rotations	Stacking ensemble (MLP + SVM fusion)	Malware detection

In the paradigm of manifest component-oriented studies, the major trade-off is that between structural expressiveness and scalability static fragility: more ICC- or semantics-conscious modeling can be done, but it requires a correct analysis of the result and results in more analysis cost. Two primary design lines can be identified in the evidence table (n = 3): ICC-semantic formal modeling (1/3) and ML over manifest-centric features (2/3), informally consisting of feature ranking selection and, in certain instances, multi-signal fusion. Formal ICC-semantic methods are capable of reasoning about interaction patterns including collusion but are susceptible to non-observable or hidden communication patterns and may not

perform well when behavioral construction is done dynamically. Manifest-centric ML pipelines can be more readily scaled by ranking and filtering high-dimensional attributes like permissions and intent filters but their performance tends to be confounded by the complexity of apps and overlapping with secondary signals in the case of fused permissions, APIs-flows. The reported performance is different since manifest indicators are dependent on configuration granularity and static visibility exported components and permissive intent-filters are informative, but dynamic loading and hidden flows may result in missed detection, and unreliable results. Thus, comparability relies on the articulation of the modeled component/ICC scope and the scope of the modeled static assumptions, recording the feature selection fusion decisions, and matching the evaluation protocols with the proposed threat model collusion targeted.

3.5 Metadata and Certificate-Based Static Detection

Metadata and certificate-based detection strategies remove the analytical focus from code execution and instead put the focus on the structural and descriptive attributes of an application. By analyzing the digital signatures, the marketplace information, and the package identifiers, or by converting these elements into different modalities, researchers have devised efficient ways of identifying malicious families and variants. A growing body of research is looking into translating non-code components into visual representations for analysis. Beyond the internal file structures, there is external data from the ecosystem that acts as a rich source of detection signals. One study used metadata of apps from the store and proposed a new labeling criterion, application lifespan defined as Potentially Harmful Apps (PHAs), which have been removed from the market within a timeframe of one month [32]. By using Random Forest classifier trained on 601 features extracted

from market metadata, the accuracy of the model was 90% which proved that the survival time in the marketplace is a less biased signal to identify the malicious campaign than traditional antivirus labels [32]. Taking a different approach to the external data, MalVulDroid uses Natural Language Processing (NLP) to analyze reports of textual description from security vendors instead of the APKs themselves [33]. By mapping these natural language descriptions to the vulnerability categories using Multilayer Perceptron, the framework succeeds in mapping the software vulnerabilities exploited by certain malware families to enrich static analysis with external threat intelligence [33].

In order to model complex relationships between a variety of static attributes and advanced neural architectures have been applied to metadata. The Trandroid system uses Transformer-based architecture for modeling the interactions between heterogeneous features, such as permissions, components, metadata attributes, and family labels [34]. Tested on the benchmark dataset TUANDROMD, the accuracy of the Transformer model reached 99.25%, which proved that the self-attention mechanism is more powerful than the traditional CNNs or RNNs in identifying the hidden long-range dependencies in hybrid static feature datasets [34]. Finally, the efficiency rapid signature distribution of the detection are critical for the large-scale detection. The FLSH framework replaces the bulky machine learning models with similarity hashing by using ssdeep to generate context-triggered piecewise hashes of important static artifacts [35]. This way it becomes possible to calculate a Net Similarity Index to detect variants with an accuracy of 96.67% against even minor obfuscations to cryptographic hashes [35]. In order to get final thought of this approach, Table VI represents contributions of each article related to this approach.

Table VI: Article contributions in metadata and certificate-based

Study ID	Year	Primary metadata / certificate signal	Representation / feature engineering	Detection / model core	Task
[33]	2022	External security-vendor text reports (metadata about malware)	NLP: BoW + n-grams (1–4) + TF-IDF, mapping families ↔ 9 vulnerability categories	MLP (best), SVM, RIDOR, PART	Vulnerability-category prediction (capability/ vulnerability analytics)
[32]	2021	App-store metadata + lifespan labeling (removed <1 month vs survives >6 months + 0 VT detections)	601 normalized static features from manifest + market metadata (permissions, hardware, category, etc.)	RF (best), SVM, SGD, XGBoost	PHA detection for app-market screening
[34]	2025	Mixed manifest + metadata attributes within a specialized benchmark (TUANDROMD)	Reverse engineering + feature encoding into numeric vectors, transformer models interactions among heterogeneous static features	Transformer (vs CNN/LSTM/GRU/ RNN/CNN-LSTM baselines)	Malware detection (and family/variant context in benchmark)
[35]	2025	Fuzzy hashing (ssdeep) over unpacked static artifacts (manifest/dex/ structural sections)	Generate ssdeep hashes, compute Net Similarity Index vs malware fingerprint repository, manual refinement to reduce noisy sections	Similarity-hash indexing (lightweight, not heavy ML)	Binary detection + variant/family identification

According to literature, metadata certificate-based investigating the trade-off between ecosystem sensitivity and generalization is the most common option: such indicators may be operationally helpful, but their predictive ability will change with marketplace policies, the extent of regional exposure, and temporal decadence. In this body of evidence (n = 4), four design directions can be identified that are, market-level lifespan proxies, vendor-text NLP, transformer modeling over heterogeneous features such as metadata, and fuzzy-hash similarity to identify variants, with each study using one design direction (1/4). Since these instructions

are aimed at different goals triage vs semantic intelligence vs family-variant matching vs detection, the concept of accuracy cannot be interpreted unanimously and may result into contradictory impressions when compared as one detection measure. In addition, features facing the ecosystem can easily be manipulated by presentations and policy-driven changes, and certificate or marketplace identities can be rotated or repackaged to minimize transferability over time and space. Thus, the ecosystem context (market/timeframe/region), the task objective, and the labeling assumption should be clearly reported by cross-study synthesis instead of comparing the results through the accuracy.

3.6 Hybrid Static Feature-Based Detection

Hybrid static detection is the amalgamation of static indicators as diverse as permissions, API calls, opcodes, control flow graphs, and metadata into unified frameworks. By synthesizing these heterogeneous sources of data, researchers hope to overcome the limitations of single-view analysis, such as susceptibility to obfuscation and lack of semantic context. Researchers have built intricate neural architectures to represent intricate interactions in hybrid static features. HDNFDroid [36] uses Stacked Denoising Autoencoders to learn features from the sparse data and then fine tunes with a supervised learning algorithm. Moving towards efficiency, BL-AMD in [37] uses Broad Learning Systems to obtain high accuracy without heavy computation. However, study [38] use Residual LSTMs and Cascade LSTM-GRU networks to capture long term dependences in feature sequence. Furthermore, methods such as study [39] support that combining different features like opcodes, strings, shared libraries, and APIs in Multi-Layer Perceptron and Deep Neural Networks is consistently better than single feature baselines. Beyond ordinary deep learning, there are a number of studies that deal with the interaction of the features.

The framework in [40] adds a hierarchical detection approach that combines block-level, module-level and app-level features, by which it finds that authorization-sensitive features which are very discriminative. Similarly, addressing the sparsity of static data, factorization machines are used to explicitly model pairwise feature interactions [41], authors use FP-Growth to mine frequent co-existence patterns of permissions and APIs, proving that feature combinations are more powerful than isolated attributes. In addition, transforming code into visual or graph structures enables the application of computer vision and graph algorithms. LensDroid in [42] combines three views API graphs using GCN, Opcode sequences using TextCNN, and structural artifacts using ConvNeXt with multi-modal bilinear pooling. However, MalFlows in [43] models apps as Heterogeneous Information Networks (HIN) using flow2vec embedding for capturing semantic flow. In addition, visual approaches include DCEL in [44], which combines 1D vectors with 2D images in order to parallelize DNN and CNN processing while study [45] uses Local Binary Patterns (LBP) in order to code images. In addition, [46] introduces a pipeline that transforms BERT-embedded feature tokens into images for CNN classification and SARVOTAM. In order to handle high dimensionality, nature-inspired optimization is widely used. The Enhanced Parrot Optimizer in [47] and Artificial Bee Colony (ABC) in [48] are used to select the optimal subsets of features hence improving the classifier performance significantly. Furthermore, for ransomware specifically a Binary Particle Swarm Optimization (BPSO) scheme in [49] combined with SMote oversampling resolves the issue of class imbalance well.

Robustness can often be accomplished through ensembles. ANDFRF in [50] proposes a fuzzy rank-based ensemble for managing classifier uncertainty, whereas JDroid in [51] is a stacked ensemble for Dalvik opcodes and Java bytecode. Scaling this to massive datasets, in [52], AdaBoost is used over 56,000 static attributes

extracted from half a million apps and with high stability. Going back to the issue of lack of data and labeling, the Siamese one-shot learning framework in [53] identifies malware families with very few samples. MADS in [54] uses light-weight rules and type-specific Random Forests to prevent installations in real-time. Moreover, PAIRED in [55] works on explainability using SHAP with a light Random Forest and study [56] shows that a shallow Random Forest with selected features is possible for ultra-fast detection (<100ms) suitable for mobile devices. Hybrid methods are also specific to a particular situation. The article [57] identifies financial fraud based on features in the domain such as package naming patterns. Also, the research [58] focuses on ransomware by using a combination of URL lexical analysis and API indicators. Finally, critical studies such as [59] put the spotlight on the importance of reproducibility and demonstrate that tree-based models are often more than simple neural networks can perform when the confounders from the experiment are controlled. Pushing boundaries even further, [60] suggests a multimodal framework that integrates static artifacts with dynamic runtime traces, making a new link between static and dynamic analysis. Transfer learning is also investigated in [61] to enhance the generalization to zero-day variants with the help of large scale offline features. In order to get final thought of this approach, Table VII which represents contributions of each article related to this approach.

Table VII: Article contributions in Hybrid-based

Study ID	Year	Hybrid static signals used	Core representation / idea	Model / learner	Task
[36]	2022	APIs + permissions + dataflow	Deep feature learning for sparse/high-dim static (MSAE+SDAE, unsupervised + supervised fine-tune)	Hybrid deep (autoencoders + classifier)	Detection
[40]	2021	Basic blocks/bytecode + permissions + authorization-sensitive APIs	Hierarchical fusion (block → module → app)	Hierarchical classifier	Detection
[37]	2021	Permissions + intent actions + sensitive APIs + etc.	Broad Learning (low compute, static-only)	Broad Learning–based detector	Detection
[54]	2023	Manifest + metadata + limited static features (no decompiling)	Rule-based prefilter → type-specific RF detectors (multi-stage)	Rules + Random Forest (per type)	Type-specific detection
[41]	2023	Permissions + API calls (+ frequent-API by frequency)	FP-Growth top-k co-existence itemsets (size 2–5) → binary combo features	RF/J48/SVM/ KNN/LR	Detection
[49]	2021	Permissions + API calls (ransomware focus)	SMOTE + wrapper BPSO feature selection	RF/SVM/DT/ KNN	Ransomware detection (imbalanced)
[44]	2024	Permissions + APIs + intents + command/ signature features	Chi-square selection → dual encoding: 1D vectors + 2D images → vote fusion	DNN + CNN + majority vote	Detection
[53]	2023	Permissions + APIs + strings	Siamese CNN learns similarity, N-way one-shot	Siamese CNN (one-shot)	Detection + family classification
[52]	2022	7 groups (perms, components, method tags, intents, packages, APIs, services/receivers)	56k binary attributes, AdaBoost over classical ML	SVM/DT/ KNN/NB/RF + AdaBoost	Detection
[55]	2022	Drebin-215 hybrid features	RFE reduces 215→35, RF + SHAP explainability	Random Forest + SHAP	Detection
[39]	2025	Permissions + intents + API calls	Hybrid DNN, removes leakage-prone metadata, cross-dataset eval	DNN (binary + family layer)	Detection + family/ category

Study ID	Year	Hybrid static signals used	Core representation / idea	Model / learner	Task
[45]	2025	Manifest + Dex bytes (image-based)	Byte streams→grayscale, LBP features, CNN ensemble (VGG/ResNet/DenseNet) + focal loss	CNN ensemble	Detection
[57]	2025	71 hybrid static features incl. package-name stats + source + user activity features	Daily retraining simulation (92 daily datasets), boosted trees	LightGBM best (plus XGB/ CatBoost/RF/ LR)	Financial malware detection
[50]	2025	Hybrid static vectors (perm + API indicators + others)	Fuzzy rank-based fusion of probabilities	KNN/SVM/LR/ XGB/LGBM + fuzzy fusion	Detection
[38]	2025	Permissions + API-related features	RF feature importance + Cascade LSTM–GRU, EOA hyperparameter optimization	Cascade LSTM/GRU + EOA	Detection
[47]	2025	214 perms + 27 API calls (241 attrs)	PO/PSO/GWO/POPSO + SA refinement for feature selection	DT/GB/HGB/ RF/XGB	Detection
[48]	2025	Drebin-derived 216 static features	ABC feature selection + RF/ XGB	RF, XGBoost	Detection
[58]	2025	URL lexical + API-call indicators (ransomware)	Huge URL feature set + stacked ensemble (RF/ LGBM → MLP meta)	Stacked ensemble	Ransomware detection
[51]	2025	SMALI opcodes + Java bytecode (hybrid opcode)	ExtraTree FS to 461 features, stacking ensemble	Stacking ensemble	Detection
[43]	2025	Control-flow + dataflow + ICC (no execution)	HIN + meta-paths, flow2vec embeddings, channel-attention DNN + MLP	Attention DNN + MLP	Detection
[56]	2025	Permissions + sensitive APIs + components/ intents	Multi-stage FS (IG + Chi² + RFE) → ~10–20 feats, shallow RF emphasized	Shallow RF (+ LR/NB/DT)	On-device / low-resource detection
[61]	2025	Broad static mix (manifest + metadata + APIs + opcodes)	46,648→10,523 selected, transfer learning, multi-class families, releases SAMDroid	DNN/CNN + transfer	Multi-class family identification
[59]	2024	Drebin-style hybrid vectors + APIGraph clustered APIs	Rigorous protocol: dedup vs duplicated, equal tuning budget, offline vs monthly active learning	RF/SVM/XGB/ MLP/SCC/HCC	Detection under reproducible settings

Study ID	Year	Hybrid static signals used	Core representation / idea	Model / learner	Task
[60]	2024	Static + dynamic (multimodal)	Train per-view models and fuse outputs	Fused supervised models	Detection
[46]	2024	Permissions + intents + receivers + services → BERT embeddings → image	Tokens→BERT mean pooling→grayscale images, CNN, Optuna tuning	Custom CNN (+ backbones compared)	Detection
[42]	2025	API callgraph + opcode grams + artifact bytes (dex/xml/so)	3-view embeddings (GCN + TextCNN + ConvNeXt) → MFB pooling + self-attention fusion	Multi-view deep fusion	Detection

The hybrid models themselves are not superior; only in cases when the fused modalities are complementary, the fusion strategy fits the threat model and is evaluated in leakage-resistant protocol, the gains are obtained. The most common backbone in the hybrid evidence table is APIs (19/26), then there are permissions (17/26), and manifest component exposure signals are also represented in a smaller, but significant proportion (10/26). The trend signifies that the majority of hybrids purposely integrate announced capability of permissions with carried out conduct of APIs and, by some counts, exposure entry points constituents. There are three repeating fusion recipes that are observable in the literature. First, the most common recipe is to combine APIs and permissions (16/26) with the ability to use capability-behavior complementarity to reduce over-declared-permission false positive and promote discrimination. Second, the APIs permissions recipe can be commonly scaled up with manifest component exposure (6/26) to include entry points and attack surface and capability and behavior. Third, it can be effective with broader feature-bank hybrids (3/26) where modalities are still complementary, though more susceptible to redundancy and dataset-specific artifact and so demand a well-thought-out fusion design and validation controls. A different sub-trend

provides extra low-level views or structure (4/26) to represent higher-order dependencies, and better deal with sparsity, at the expense of more complex extraction, and sensitivity to incomplete static evidence. On the other hand, the failure mode of hybridization occurs when fusion only raises dimensionality and provides near ceiling results that are not applicable across datasets, time or protocols.

3.7 Adversarial-Aware and Robustness-Oriented Detection Models

As machine learning becomes a standard for malware detection research has shifted toward understanding and countering adversarial attacks. This is the area of vulnerability analysis, feature hardening, adversarial training, and novel architectural defenses. Several studies have revealed the fragility of current ML detectors. To quantify this threat, researchers created Attack Effectiveness Classifiers (AEC), which predict the probability of success of specific evasion techniques [62]. Experiments demonstrated that problem-based evasion attacks that exploit obfuscation and flow manipulation could lower detection rates in commercial scanners by up to 97% [62]. Furthermore, the OFEI framework proposed the semi-black-box attack via Simulated Annealing that achieved the misclassification rate of 98.25% against Deep Learning as a service (DLaaS) platforms using modification of only one feature per iteration [63]. Talking about the obfuscation of code directly, the tool Deoptfuscator is concerned with bringing apps obfuscated by DexGuard back to a High-Level Intermediate Representation [64]. It was able to recover obfuscated apps with an average similarity of 0.93 to the original apps and thus allows standard static detectors to work correctly [64]. One of the big trends in the field of defense is to include adversarial examples during the training process. Theoretical guarantees were added by the PAD framework, which involves Input

Convex Neural Networks (ICNN) and Stepwise Mixture of Attacks to solve robust training as a constrained optimization problem [65]. PAD was able to maintain more than 83.45% accuracy under attack which beats heuristic defenses [65].

Going just beyond robust classification, some articles try to detect the attack attempts themselves. MalProtect is a stateful defense layer where it keeps the history of up to 10,000 queries to detect abnormal patterns in indication of a probing attack [66]. It reduces evasion rates by more than 80% for major benchmarks [66]. Complementing this, the defense mechanism that comes with OFEI uses Bayesian Neural Networks to estimate aleatoric and epistemic uncertainty [63]. This approach identifies adversarial inputs based on the clusters in the feature space, for which the uncertainty is high. 99.28% detection rate [63]. Finally, complex representations of data are being capitalized on in order to improve resilience. A graph-based framework based on Graph Neural Networks (GNNs) on API graphs used a Variational Graph Autoencoder (VGAE) and GAN to obtain adversarial graphs for training [67]. This adversarial-aware GNN showed an accuracy of 98.33% on CICMalDroid [67]. Another method presented a dual-modal detector based on a spatial (ResNet-50) and frequency domain feature of the DEX code [68]. Using random erasing as a training technique to simulate the obfuscation process, this model achieved 97.3% accuracy on Drebin and proved to have a higher robustness against FGSM attacks than single-modal baselines [68]. In order to get final thought of this approach, Table VIII represents contributions of each article related to this approach.

Table VIII: Article contributions in Adversarial Malware Detection Approach

Study ID	Year	Role	Target detector / feature space	Attack / defense mechanism	Model family
[62]	2022	Attack + Meta-modeling	Replicated SVM/FM/ DNN detectors trained on Drebin, benign from AndroZoo, tested also vs VT engines	Problem-based code-modifying evasions (camouflage/ obfuscation/noise/flow manipulation), AEC models predict attack success per detector	Attack suite + effective predictors
[64]	2022	Robustness enabler (DE obfuscation)	DexGuard level 3 obfuscated apps, improves downstream static analysis	Deoptfuscator: DEX→HIR, remove opaque predicates/ global opaque vars, uses ReDex for L2, rebuild executable APK	Static DE obfuscation tool
[67]	2023	Attack + Defense (adversarial-aware graph)	API graphs + permissions + intents, GNN embeddings + classifiers	Build local/global API graphs, train GCN/GraphSAGE, VGAE + GAN generate adversarial graphs (VGAE-MalGAN), adversarial training improves	GNNs + GAN/ VGAE + downstream classifiers
[66]	2023	Defense layer (query attack detection)	Feature-based detectors (Android+Windows) on AndroZoo/ Drebin/SLEIPNIR	MalProtect front-end: query history (10k), indicators (L0 distance, shared feats, count stats, AE loss) → LR/NN to detect attacks, blocks/overrides output	LR / small NN (guard)
[63]	2024	Attack + Defense	DLaaS deep detectors on static binary vectors	OFEI: one-feature-per-iteration, simulated annealing guided by output scores, Defense: Bayesian NN uncertainty (aleatoric+epistemic) detects adversarial inputs	Attack + Bayesian defense
[65]	2024	Defense (principled adversarial training)	DNN on rich Drebin-like hybrid static features	PAD: ICNN adversary detector + constrained robust optimization, PAD-SMA mixture of attacks (FGSM/BGA/BCA/Grosse/JSMA/ Mimicry) during training	DNN + ICNN
[68]	2025	Defense-oriented robust detector (augmentation + fusion)	DEX bytes → spatial fingerprint + frequency spectrum	CNN (ResNet-50) + spectral features (2D FFT) → DFN with attention + recursive optimization, random erasing to mimic occlusion, tested under FGSM	CNN + fusion

The existence of robust claims in adversarial-conscious malware detection needs a clear threat model; otherwise, apparent accuracy cannot be generalized to attackers as to their objectives, information, and capabilities. In this regard, the research ought to specify the objective of the attacker, the knowledge regime such as white and black-boxes, the capabilities, and limitations, in line with the common adversarial-ML practice. The strength of robustness is explicitly viewed as resistance to attacks in most studies in this evidence set (6/7), whereas more explicit transformations of obfuscation are mentioned in a smaller fraction (2/7). Assumptions on the knowledge of the attackers also differ explicit black-box and query-driven environments are found in (2/7), explicit white-box gradient testing in (2/7), and the rest are based on mixed or implicit regimes. Defenses have been found in adversarial training, attack mixing, guard layers indicating probing by query pattern or uncertainty, code-level recovery (deobfuscation) to stabilize downstream static signals, and robustness-by-design. The often-discussed mismatched threat model and mismatched evaluation budget can also be understood as resulting in conflicting high-robustness claims: What was earlier read as robustness like clean accuracy,and low level of computation might be lower when under adaptive query-efficient attack.

3.8 Graph-Based and Structural Static Detection

Graph-based detectors model applications using explicit program graphs such as control-flow graphs (CFGs), call graphs, or inter-component communication (ICC) graphs. Their main bottlenecks are graph extraction cost, scalability on large corpora, and graph fragility under obfuscation and reflection, which can lead to missing edges or nodes and degraded generalization. To overcome the limitations of bag-of-words static analysis, which often fails to capture execution logic and

obfuscated relationships, researchers have increasingly turned to structural representations. These approaches treat Android applications as complex graphs (Control Flow Graphs - CFGs, Data Flow Graphs - DFGs, Function Call Graphs - FCGs) or transform code into visual and mathematical structures suitable for advanced deep learning. A major trend involves leveraging Graph Neural Networks (GNNs) to learn directly from code structure. The AdMat framework in [69] transforms API-call graphs into adjacency matrices, effectively treating them as images for CNN processing, achieving 98.26% accuracy by bridging graph semantics with computer vision. Similarly, HAWK in [70] model's applications as Heterogeneous Information Networks (HINs) connecting APIs, permissions, and libraries. It utilizes a Meta-Structure Guided Graph Attention Network (MSGAT) to learn robust embeddings, capable of detecting out-of-sample apps in milliseconds. At the application level, another framework in [71] builds a global similarity graph of apps and applies GraphSAGE with an MLSTM aggregator, significantly improving robustness against adversarial attacks like FGSM. Several studies abstract code into topological structures to analyze behavior without heavy semantic parsing. SNADroid in [72] models function call graphs as social networks, ranking structural metrics like Katz Centrality and Triangles to identify 15 highly discriminative features. Transforming bytecode into images allows the application of powerful pre-trained vision models. The study [73] converts DEX bytecode into color and grayscale images, fine-tuning models like Xception and ResNet to achieve 99.40% accuracy. SAC in [74] advances this by fusing content and structure. It treats DEX content as RGB images and bytecode instructions as graphs, outperforming both N-gram and standard graph baselines. For ransomware specifically, the framework in [75] combines ResNet-18 visual features with fuzzy hashing (ssdeep), ensuring robustness against minor code obfuscations. Another approach in [76] employs Stacked Autoencoders (SAE) to

compress high-dimensional binary images of APKs before CNN classification, enhancing computational efficiency. Deep structural analysis addresses specific blind spots in static detection. LibRoad in [77] and LibCapsule in [78] focus on Third-Party Libraries (TPLs). LibRoad uses structural signatures to accurately detect libraries, while LibCapsule enforces isolation policies on untrusted library code. Addressing native code risks, uDep in [79] uses mutation-based analysis to generate dependency stubs for JNI functions, significantly improving taint analysis precision. Finally, DBank [80] constructs a Triadic Suspicion Graph specifically for Banking Trojans, using PageRank-like scoring to detect fraud with 99.9% accuracy. Other novel approaches include MMPD in [81] which analyzes PDF structure labels for mobile robot security, and the framework in [82] which treats memory dumps as static big-data artifacts, processing them with Apache Spark for forensic detection. In order to get final thought of this approach, Table IX represents contributions of each article related to this approach.

Table IX: Studies summary in Graph-Based Approach

Study ID	Year	Structural / graph signal	Representation / feature engineering	Model / analysis core	Task
[69]	2021	API-call graph (nodes=APIs, edges=relations)	Graph → 219×219 adjacency matrix image, API space reduced (~23k→~17k)	CNN	Detection + family classification
[72]	2024	Function call graph as social network	57 graph metrics, rank via T-test/NMI/MIC → ~15 top features	RF/DT/1NN/3NN	Detection
[73]	2022	Bytecode as images (classes.dex → PNG)	Color/grayscale images, transfer learning across 16 CNNs	Fine-tuned CNNs (ImageNet TL)	Detection
[75]	2023	Bytecode images + fuzzy hash similarity	DEX→RGB + ResNet-18 embeddings, ssdeep similarity scores, fuse	RF/SVM/KNN/LR	Ransomware detection
[76]	2023	Raw APK → binary image/vector	SAE compresses binary images → latent, CNN on compressed map	SAE + CNN	Detection

Study ID	Year	Structural / graph signal	Representation / feature engineering	Model / analysis core	Task
[80]	2021	Triadic Suspicion Graph (apps–APIs)	PageRank-like suspicion scores/ ranks, fused with lightweight static+dynamic	Graph scoring + ML	ABT detection (multi-class: goodware/ ABT/ other)
[70]	2024	Heterogeneous Information Network (App–API–Perm–Class–Interface)	Meta-path/meta-graph + MSGAT, incremental embedding MSGAT	Graph Attention + RF/LR/SVM	Detection
[78]	2022	Structural confinement of third-party libraries	Static rewriting + instrumentation (incl. dyn dex + native isolation) enforcing XML policies	Enforcement framework	Privacy/ security enforcement
[77]	2022	Structural signatures for TPL identification	Weighted signatures over classes/packages, name+similarity matching, obfuscation-aware filters	Library detection tool	TPL detection
[82]	2022	Memory dump structure (RAM artifacts)	Forensic features (processes/ DLLs/registry traces) in Spark	LR/GBT/RF/LSTM	Detection
[81]	2022	PDF internal object structure	Structural tokens (/JS etc.) + TF-IDF-like weighting + FS	SVM/DT/KNN/ DNN	Malicious PDF detection
[79]	2023	JNI-aware dependency graph for taint analysis	Mutation-based dependency inference on native funcs, generate Jimple stubs for DroidSafe	Static taint analysis enhancement	Precision/ coverage of taint analysis
[71]	2025	App–app similarity graph (homogeneous)	Features (perms/APIs/TPLs) create edges, GraphSAGE with MLSTM aggregator, chi-square FS	GraphSAGE (MLSTM)	Detection
[74]	2025	Dual: DEX content image + bytecode structure graphs	Content: CLAHE RGB + ConvNeXt-style IFNeXt, Structure: Word2Vec + dual graphs (GI, GII) + GAT/GCN, fuse → MLP	CNN + GAT/GCN fusion	Detection

In both graph-based and structural static detection, the most important trade-off is between the fidelity of graphs and the interpretability of the higher-order relations they can represent more detailed program-structure graphs can represent more fidelity, but the resulting representations and the decisions learned are more difficult to interpret and audit compared to compact representations. The literature in the evidence table (n = 14) is not assembled around one graph recipe, but instead it is divided into an even (7/14) of explicit graph construction and structure proxies that are not based on whole-graph reasoning (7/14). Recurrent families are also known among explicit graphs, e.g. Call-structure graphs like API-call or function-call relation, relational graphs over heterogeneous entities like HINs and app-app similarity graphs, suspicion graphs, and dependency taint-oriented variants including JNI-aware dependency graphs. The proxy structural encodings graph images and bytecode images are better scalable and pipeline-friendly and are compromised in detailed control dataflow meaning. The extraction assumptions and noise are the two principal factors contributing to the appearance of conflicting results: reflection and dynamic loading may eliminate or corrupt edges, nodes, and third-party libraries might introduce third parties to the graph that drown the malware-specific patterns.

3.9 Transformer-Based Static Detection Models

In transformer-based static detection, tokens typically correspond to API call sequences, opcode-bytecode tokens, or graph-derived walks substructures linearized into sequences. For fair evidence-based comparison, transformer results should be discussed against strong CNN/RNN baselines under matched splits and dataset conditions, because gains may otherwise reflect data label effects rather than architecture advantages. Current research into static malware detection is, for

the most part, veering away from using conventional and recurrent neural architectures and moving toward models that incorporate self-attention mechanisms and generative artificial intelligence. Such a transition supports the rendering of long-range dependencies in the source codes, and allows the study of binary semantics in the same way as that of natural language processing. The SHERLOCK framework in [83] is a fit example of such a trend. By mapping DEX bytecode into visual representations and using Masked Auto-Encoder (MAE) in a self-supervised learning regime over 1.2 million unlabeled Android applications, SHERLOCK detects the structural grammar of malware without using labelled data. Subsequent supervised fine tuning results in detection accuracy of 97 offering resilience to code obfuscation and shows the model's advantages of scalability compared to traditional convolutional networks. Transitioning from image based to text-based representation, SecretLoc in [84] is the first large language model (LLM) application for static code audit which uses GPT-4o-mini. Unlike conventional detection of hard coded secrets using regular expressions, SecretLoc uses the contextual reasoning capability of the model to interrogate the surrounding code. Consequently, the system recovered ~93% of previously documented secrets, and found some additional, obfuscated credential formats that evaded the texture of discussed tools, nominally asserting the utility of generative AI in static security assessment. In the field of feature-based malware detection, IPR – EODL in [85] extends the classical long - short - term memory (LSTM) network using a channel attention module. The attention mechanism places higher weights based on discriminative features based on how this relates to security outcomes, while the LSTM component captures the temporal dynamics of those features. Parameter optimization is performed through the Equilibrium Optimizer (EO) in which after optimization the obtained system has an accuracy of 99.18, thus proving the effectiveness of attention in improving sequential models for static feature vectors.To complement these

neural approaches are sophisticated signal processing techniques. SpecView in [86] uses Singular Spectrum Transformation (SST) to analyze binary executables as 1D time series signals to find change points that would imply packing or encryption. This spectral signature method was 100% for highly obfuscated families where image-based methods failed. By preserving spatial locality, this approach makes it possible for lightweight CNNs such as LeNet to detect the presence of ransomware in native binaries with 99.7% accuracy and high energy efficiency approximate 1 Joule per scan, which is ideal for mobile deployment. In order to get final thought of this approach, Table X represents contributions of each article related to this approach.

Table X: Studies in Transformer Approach

Study ID	Year	Transformer-based	Static signal	Representation / key idea	Model core	Task
[85]	2024	ViT	DEX bytecode → byteplot images	Self-supervised MAE pretraining with ViT (75% masking) on MalNet ~1.2M apps, fine-tune for downstream tasks	ViT/MAE + small head	Binary detection + type (47) + family (696)
[84]	2025	LLM	Hardcoded secrets (strings + code context)	Decompile XML + code strings, candidate mining → LLM labeling (secret type vs NOT-SECRET) for context-aware auditing	LLM-assisted static auditor	Secret discovery (security auditing)
[83]	2022	Attention-LSTM	Permissions + API calls	Binary vectors, Channel-Attention LSTM (CA-LSTM) learns feature weights + dependencies, Equilibrium Optimizer tunes hyperparams	CA-LSTM + EO	Malware detection
[86]	2021	Signal+ML	Raw binary stream as 1D signal (PE + APK)	8-bit time-series → Singular Spectrum Transformation CP scores, PSO tunes SST parameters, ML ensemble on CP vectors	RF/ GNB/ET + Voting	Detection + packed/ encrypted robustness

In transformer-oriented detecting static transformation, the main trade-off across transformer-non transformer is not tokenization modality decision, but comparability vs. semantic continuity represented: it depends on what limited list counts as a token to determine the quality of cross-study comparison. In such an evidence set (n = 4), tokenization is modality-dependent and has four directions, including vision-based tokenizers on top of bytecode images (1/4), LLM-based auditing on text (1/4), enhancing sequence models with attention on sparsely indicative signals (1/4), and segmenting raw binaries with signals (1/4). Since the directions are also aimed at fulfilling various goals and naming spaces, such as auditing vs detection, and binary vs multi-class family, the values of headline accuracy are not directly comparable, even in a case where there are self-attention models. Other conflicting assertions are usually motivated by unsurpassable bases and information conditions: increase could be dependent on pretraining weight or on the quality of labels or label splits instead of attention itself.

4. Datasets Used in Static Malware Detection

It is imperative to possess a comprehensive understanding of the numerous datasets that are employed in Android malware detection in order to assess the generalizability, reproducibility, and real-world applicability of detection algorithms. This is due to the fact that these datasets are employed to detect malicious software.

4.1 Public Benchmark Datasets

The researchers employed a diverse array of datasets in the 78 papers that were analyzed, six popular datasets mentioned repeatedly. These datasets are as the following:

1- **Drebin**: Drebin is an Android malware benchmark used as a classical example to publish reproducible research on the topic of detecting malware when it is in static form. The dataset available on the official Drebin site is 5,560 malicious Android applications (APKs) representing 179 malware families, which were collected in the time period between August 2010 and October 2012, and samples were provided to the authors by the MobileSandbox project, so the access is limited and credentials are provided after requesting them by email to prevent abuse [23], [26],[27].

2- **AMD (Android Malware Dataset**): AMD (Android Malware Dataset, the Argus Lab) is an Android malware-only corpus, built to provide ground-truth family/behavior semantics beyond what simple AV labels provide: the technical report describing the creation of the corpus describes how 24,650 malware APKs were grouped into 71 malware families, and the families further clustered into 135 behaviorally distinct subgroups behavioral profiles reports of which are provided in the report, samples were gathered on VirusShare, Google Play, and third-party [18][26],[33].

3- **Malware Genome Project (MalGenome**): The Android Malware Genome Project dataset, also known as the Malware Genome Project (MalGenome) or Android Malware Genome Project dataset, is one of the early malware-only corpus that was extensively reused in subsequent studies: a manually/automatically crawled collection of 1,260 malicious APKs in 49 malware families, purported to represent the majority of known Android malware between August 2010 and October 2011, gathered by Zhou and Jiang to use in their 2012 SoK study: MalGenome practically is most frequently employed in malware-family analysis/

classification and baseline detection projects, however, researchers typically must combine it with a distinct set of benign apps (MalGenome itself is malware-only) and it is very skewed in its distribution of families. [41][44][55]

4- **CICMalDroid 2020**: The dataset is a hybrid of a static dataset and a dynamic dataset, which is created and trained to classify malware categories: CICMalDroid2020 (Canadian Institute for Cybersecurity, UNB) consists of 17,341 Android apps (APKs) gathered using a variety of sources (VirusTotal, Contagio, AMD, MalDozer and other sources mentioned in the article by the authors) between 2017 and 2018 and is categorized into 5 classes (Adware, Banking malware, SMS malware, The developers have dynamically tested the APKs with CopperDroid and state that 13,077 apps were successfully launched; after eliminating problems like failures in JSON parsing. The number of apps that can be used is 11,598 apps (Adware 1,253; Banking 2,100; SMS 3,904; Riskware 2,546; Benign 1,795). It contains (1) APK files (17341), (2) logs of dynamic analysis captured (13077), and (3) various CSV views of features such as 470 dynamic features like system calls, binders, and composite behaviors. The dataset has 139 system-call-only features, a 50621-feature static set like intents, permissions, services, sensitive APIs. Also, as well as PCAP network-traffic captures of the actual runs; the dataset page also presents the important papers [55],[46],[67].

5- **AndroZoo**: AndroZoo is an academic, ever-growing collection of Android APKs, so far maintained by the University of Luxembourg. The official portal reports 26,694,440 APKs in the collection, collected by several sources such as Google Play and other markets. APKs are served raw

(APK/ZIP with DEX byte code, certificates, assets and the manifest), and each application is scanned by numerous antivirus detectors via Virus Total-style scanning, with the data set served via authenticated download API Hashed with SHA-256. An APK list hash index containing core metadata fields commonly used in malware studies is also published periodically by AndroZoo: such as dex date for compilation time, VT detection for number of engines flagging, APK size, DEX size, package name, version code, and source market. So, researchers can filter and select subsets before downloading. To access, one is required to request an API key via an institutional email and accept terms, including non-commercial use, no redistribution without permission and having the key remain secret; keys have a time limit of 6 months and are limited to 500,000 APK downloads at a time. Also, according to AndroZoo, Google Play Metadata been introduced in December 2023 and the auxiliary information is now more diverse using fields like hash/VT in addition. [66][68].

6- **CIC-MalMem-2022**: Published by the Canadian Institute of Cybersecurity (UNB). It consists of a memory-dump focused dataset that is designed to test obfuscated malware detection by memory forensics to make it closer to real-world conditions by creating dumps, but not the artifacts of the dumping process itself (it uses debug mode, such that the dumping process itself does not appear in the captured memory). It has a total of 58,596 records (29,298 benign and 29,298 malicious) and three malware super-classes, including Ransomware, Trojan horse, and Spyware. [82]. Table XI shows most popular datasets used in current literature.

Table XI: Decision-grade benchmark datasets used in static Android malware detection (2021–2025)

Dataset	Access	Timeframe / Refresh	Benign source	Malware source	Approx. size	Label type	Duplicates handling (reported)	Split strategies
Drebin	Public download	Aug 2010 – Oct 2012	Not included (paired with markets in many studies)	Malware from MobileSandbox ecosystem	5,560 apps, 179 families	Family labels	Not stated in surveyed papers	k-fold CV, random hold-out
AMD (Android Malware Dataset)	Public benchmark	2010 – 2016	Usually paired with public benign	Curated malware corpus	24,553 malwares, 71 families	Family / binary	Not stated	Random split, k-fold CV, some cross-dataset
Malware Genome Project (MalGenome)	Academic corpus	Aug 2010 – Oct 2011	Not included	Curated malware families	1,260 malwares, 49 families	Family	Not standardized across re-uses	Random split / CV in derivatives
CICMalDroid 2020	Public	Dec 2017 – Dec 2018	Included	Included (multiple categories)	17,341 samples	Category labels (Adware/ Banking/SMS / Riskware/ Benign)	Not consistently documented in surveyed papers	Random split, CV, some multi-class
AndroZoo	Request access by emailing from an institutional address; APK download via API key + SHA-256; key expires after 6 months and capped at 500,000	Continuously growing; crawling started in late 2011 and continues; the public index is updated nightly (before ~6am Luxembourg time).	Primarily app markets such as Google Play and multiple alternative markets	Malware is not a separate feed malicious apps are a subset of the collected sources and are identified via AV/VirusTotal signals (vt_detection)	Current number of APKs shown on the portal: 26,694,440	VirusTotal-based: vt_detection = number of AV engines flagging the APK, plus	duplicate handling is not guaranteed without downloading both, and repackaged apps will have different hashes.	Not prescribed by the dataset

Dataset	Access	Timeframe / Refresh	Benign source	Malware source	Approx. size	Label type	Duplicates handling (reported)	Split strategies
CIC-MalMem-2022	Public download from CIC/UNB dataset page	Released as a fixed snapshot; no rolling refresh is described on the official	Normal user behaviour memory dumps captured while running various applications (benign activity)	2,916 malware samples collected from VirusTotal	58,596 records total (29,298 benign + 29,298 malicious) with 57 memory-analysis attributes (tabular feature dataset derived from dumps)	Balanced binary (benign vs malicious) and malware type labels (Spyware, Ransomware, Trojan Horse)	No explicit duplicate policy reported	Not prescribed by the dataset

Public datasets function as empirical common ground in Android malware research because they are openly accessible, repeatedly reused, and therefore enable reproducible comparison across detection pipelines. In the surveyed literature, two resource types dominate reuse, curated malware benchmarks (Drebin, MalGenome, AMD) and large-scale APK repositories (AndroZoo) that are commonly used to construct benign sets and to scale mixed benchmarks toward more realistic app diversity. In parallel, several studies move beyond binary malware and benign discrimination by adopting multi-class category benchmarks to evaluate fine-grained threat typing [21], and by incorporating more recent public collections (CICMalDroid2020) for category-based evaluation and cross-benchmark testing [40], [83]. The benchmark landscape also includes derived feature-vector datasets (Drebin-215) and specialized public resources used for targeted objectives such as abnormal data-flow mining [31], memory-dump–based analysis under obfuscation (CIC-MalMem-2022) [82], and malware-dump collections for sourcing malicious samples [81]. Crucially, cross-study comparability can be undermined by temporal drift (older corpora vs newer app ecosystems), label noise and AV-policy differences including VirusTotal-style reports, and evaluation protocol choices (random split vs time-aware split, potential data leakage through repackaged variants or duplicate apps). These factors should be considered when interpreting unusually high accuracy claims across heterogeneous datasets and splits.

4.2 Custom /Private Datasets

Custom and private datasets are repeatedly introduced to address practical limitations of standard benchmarks, most notably recency, label confidence, domain specificity, and controlled class balance often by re-sampling public sources, re-annotating ground truth, or engineering a representation aligned with a paper's threat model. A prominent direction emphasizes scale to improve statistical power and capture long-tail behavior, [52] aggregates 500k+ apps from multiple platforms, while [57] reports an operational Fraud Detection System with 183M benign samples alongside ~12k malware

samples. Similarly, large, curated corpora are constructed in [77] (155,300 apps) and [32] (91,203 apps). A second direction emphasizes diversity of acquisition sources to broaden coverage of permission ecosystems and market practices, combining Google Play with third-party or regional stores [39], [48]. Beyond volume and diversity, several studies curate specialized datasets to evaluate threat scenarios underrepresented in general benchmarks. Examples include data-flow–centric profiling using MUDFLOW [36], and collusion-aware benchmarks designed to detect multi-app threats [19]. Other works build datasets as experiment harnesses rather than simple APK pools for instance, stressing third-party library abuse or repackaging behavior [78]. However, such custom datasets can improve ecological validity and target specific attack vectors, cross-study comparability depends strongly on clearly reporting labeling and validation procedures (multi-engine or VirusTotal-style thresholds), split strategy (random vs time-aware), and de-duplication controls, since these design choices can materially inflate or deflate headline performance. Figure (4) shows a summarization for the usage of each dataset in the collected articles.

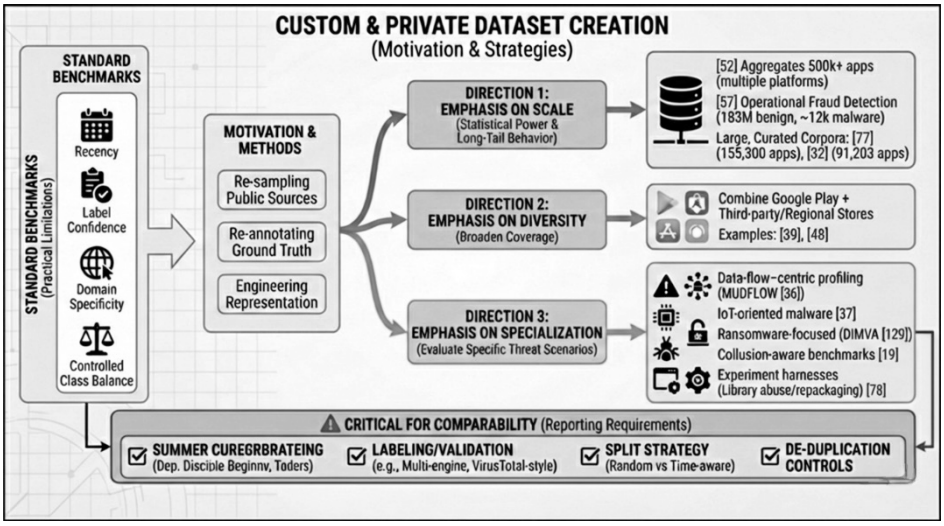

Figure (4): Custom /Private Datasets Usage

4.3 VirusTotal / VirusShare Labeled Datasets

VirusTotal and VirusShare-labeled are typically not fixed benchmarks but constructed by collecting APKs from the wild and then assigning labels using multi-engine antivirus consensus. In this category, VirusShare is mainly used as a malware sourcing repository it distributes very large malware corpora (primarily as downloadable archives torrents, and accompanying hash lists) and currently reports over 108 million samples [40]. VirusTotal, by contrast, is commonly used as a labeling verification layer because it aggregates scan results from many antivirus engines and analysis tools, allowing researchers to operationalize maliciousness through rules like detected by ≥k engines or through vendor-tag queries. VirusTotal-style labeling is inherently noisy, different AV engines may disagree on the same APK due to varying signatures, heuristics, and update cycles, and the set of participating engines can change over time. As a result, ground truth defined by a fixed threshold can be unstable across rescans and may bias evaluation toward samples that are easier to detect. Therefore, studies should report the exact threshold, the scan date, the number of engines, and when possible, provide disagreement statistics to make results comparable. This is reflected in [37] apps verified with VirusTotal, and reported to Google's Android Security Team, and [80] APKs downloaded from VirusTotal during 2016–2017 and labeled as Banking Trojans when flagged by at least five AV tools. Several papers in this set still create custom datasets but rely on VirusTotal-style verification to strengthen label credibility or to focus on a threat slice, [75] constructs a targeted dataset of 4,136 ransomware and benign samples, and [14] builds a ~16,000-app corpus with 103 permission features by filtering AndroZoo-derived data and downloading APKs HTML pages. In addition, an approach is practical because AndroZoo is explicitly designed to provide large-scale APK collections together with metadata such as VirusTotal reports. Not all

studies here are dataset-first, [29] uses a demonstration collusion case to model a realistic SMS-based contact-exfiltration path, emphasizing attack semantics more than sample volume. However, [70] describes a large-scale pipeline decompiling ~200,000 APKs. A key methodological point made explicit in [22] using 70,142 apps from 2011 to 2016 is that VT-based labels can change over time as engines update signatures and heuristics, so the same APK may shift between benign, suspicious, and malicious across re-scans, this temporal label drift is exactly why Figure (5)'s usage summary matters for interpreting comparability across VT/VirusShare-labeled studies.

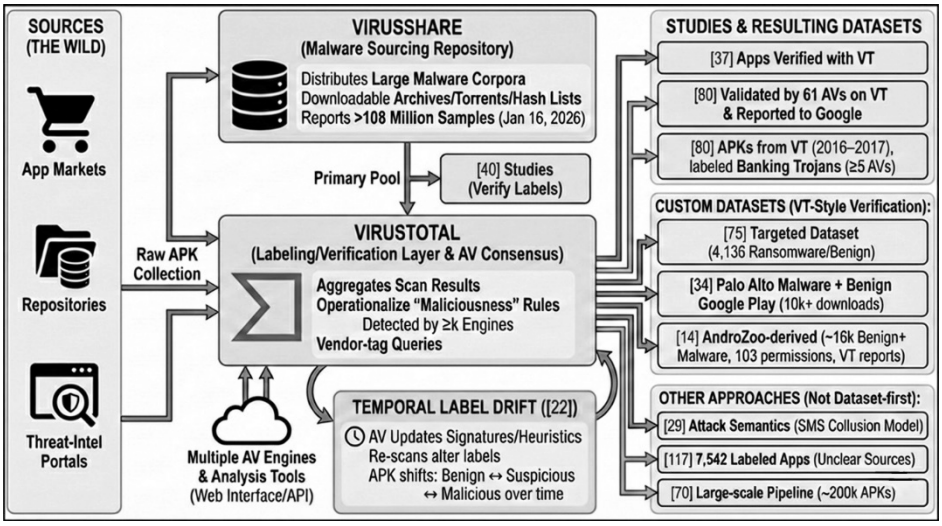

Figure (5): VirusTotal / VirusShare Labeled Datasets

5. Theoretical Gaps and Criticisms

From 2021 onwards, static Android malware detection has been researched significantly which has achieved lots of methodological advancements. However, the existing documentation still presents several theoretical and practical limitations associated with the operational deployment, long-term applicability, and the fair

comparison across different studies of such malware detection techniques. Although many investigations make use of extensive corpora and a rich set of static feature families. Additionaly, most of these studies report laboratory-scale, single-machine pipelines where scalability is seen as an incidental byproduct of the speed of classifiers rather than a fundamental requirement. Consequently, there are still a minority of studies focused on issues around distributed storage, parallel parsing, incremental updating, or reproducible orchestration, and there is no consensus within the field for a big-data reference framework, including versioned dataset management, scalable APK ingestion, multi-view feature extraction, and concept-drift evaluation. In parallel, feature engineering and selection protocols are often built ad hoc based on convenience representations such as frequency vectorization, top-k lists and single- criterion filters, which are poor for balancing accuracy with computational cost, as well as mode of interpretation, robustness to obfuscation and deployment on-device vs cloud. Moreover, there is very little utilization of multi-objective decision-making strategies - such as multi-criteria decision analysis (MCDM). These MCDM could help shed light onto the reasoning behind selecting a particular feature set that is considered optimal in the face of competing goals such as latency, memory consumption, API evolution and adversarial evasion. Adaptivity still has a long way to go, detection mechanisms are usually trained in the offline phase and tested once, despite the constant evolution of malware and android APIs. This oversight has allowed the study of sequential decision-making strategies like budgeted extraction policies and adaptive thresholding in response to varying prevalence and post-deployment feedback, but these are generally yet unexplored in reinforcement learning frameworks. Furthermore, in spite of the fact that transformer-based and inspirations are still emerging, large language models (LLMs) are rarely utilized as controlled structures for semantic abstraction like associations

of low-range artifacts to larger profiles of behavioural supplies, while at the same time prescribing explicit control on security and privacy inside the seeds. Such practices add to the gap between the representational capacity of LLMs and their implementation. Empirical data show a strong evidence of evaluation issue because 42 out of 78 studies explicitly reported split and evaluation protocols, time - split evaluation is only mentioned in one case, 53 studies did not state or the information about splits and evaluation protocols is ambiguous, 23 studies did not document sufficient information about deduplication and leakage controls - procedures that are prone to inflate performance measures if duplicates, variants, or family overlap remain across random splits. Parallel transparency issues include the construction of data, in which 17 of 78 studies have unclear provenance, 13 do not have clear labelling procedures, and 4 do not report VirusTotal (VT) or antivirus (AV) thresholding procedures. Evaluation depth is also equally limited with 11 studies not reporting on robust testing, obfuscation and adversarial evaluations carried out by just 10 studies each, cross dataset or generalization testing by 4 studies, and checks that can be performed on time can only be found in 3 studies. Moreover, measuring computational reporting, such as runtime, cost, latency and scaling constraints, is never quantified. Collectively, these deficiencies provide a good reason for a focused developmental goal, the introduction of reporting checklists including metrics for split type, deduplication, leakage mitigation, replicated run statistics, dissemination of dataset including source, temporal, labelling rules, and thresholds, the creation of a minimum robustness suite including obfuscation and repacking, cross- vs. family- vs. time- split evaluation, computationally specific metrics such as extraction time, throughput, memory footprint, and hardware requirements, and a strategic focus on scalable, multi-objective, and continuously adaptive static detection frameworks.

6. Recommendations for Future Research.

Future research must therefore move from isolated model proposals to an integrated, scalable and decision theoretical research agenda. At the level of systems, researchers should standardize a reference architecture for big-data static malware screen which explicitly provides for distributed feature extraction including the parallel unpacking, decompilation, caching of intermediate artifacts, and resilient parsing, the columnar storage Parquet-based feature lakes, and the incremental learning and evaluation over temporal slices so as to measure the aging and concept drift in the context of realistic market dynamics. Within this pipeline, feature selection should be rethought as a multi-objective design problem. Multi-criteria decision-making (MCDM) techniques, including AHP, TOPSIS, BWM, ELECTRE and PROMETHEE, can be used for scoring and selecting feature families such as permissions, semantic API clusters, certificates, graphs and metadata by simultaneously considering detection performance, extraction cost, robustness to obfuscation and API evolution, explainability and deployment constraints. The resultant output should consist of, instead of having accuracy as the only basis, an auditable rationale for selection that ties features that are selected to explicit operational criteria rather than to accuracy alone. Building on this premise, reinforcement learning, that may take the form of contextual bandits, may be plugged into the control layer of a multi-stage detector. In this setup, an agent can learn to implement policies that decide, for each application, which static analyzes to run, how to allocate computational resources at scale, when to ask for human-in-the-loop verifications, how to change decision thresholds in the face of fluctuations of label prevalence. To reduce the limitations about semantics and interpretability, large language models should be placed under constrained and privacy-aware protocols, preferably using retrieval augmented generation based on vetted security

knowledge bases. Their main functions would include the normalization and the explanation of behavioural proofs, the production of structured descriptions of capabilities based on extracted artifacts and supporting the construction and maintenance of dynamic taxonomies. Furthermore, it will be important for future research to describe the protections, such as schema-constrained outputs, resistance to adversarial prompts, and calibration mechanisms, that will mitigate hallucinations. In addition, researchers should make additional commitments to enhanced reproducibility by publicly disseminating code, feature schemas and temporally stratified evaluation splitting, reporting end-to-end throughput alongside accuracy metrics, and benchmarking robustness in standardized obfuscation, repackaging and dataset shift scenarios. Finaly, these initiatives would turn static Android malware detection from a collection of high accuracy but brittle prototypes into a scalable, explainable and perpetually intensifying ecosystem that is amenable to big data real-world deployment.

7. Conclusion

This survey was completed with a comprehensive assessment of 78 peer-reviewed studies that were carried out between 2021 and 2025. The research focused on the detection of static Android malware. The literature was organized based on the major feature modalities and modeling families used to identify the malicious programs without execution. The research describes the situations in which static detection is most effective and vulnerable, by synthesizing the extraction and modeling of permissions, the API, the opcodes, the manifest, the metadata-certificates, the hybrid features, and the deep architectures. This is especially important in cases such as obfuscation, dataset imbalance, Android API development and insufficient repeatability techniques. Although the taxonomy and dataset

analysis show that great progress is made in semantic representations and reported accuracies. The literature also shows that many solutions are tested in a small context with no scalability as end-to-end solutions. The review highlights the need for the development of novel classifiers and the development of scalable big data pipelines, multi-criteria feature selection, adaptive learning strategies, and trustworthy semantic reasoning layers, as required to shift the progression of the field. This is critical to ensure the reliability, explainability and functionality of static malware detection to adapt to evolving Android and malware ecosystems.

Conflicts of interest

The authors declare that they have no conflicts of interest.

Funding

None.

Acknowledgement

The authors would like to thank Mustansiriyah University (https://uomustansiriyah.edu.iq/) in Baghdad-Iraq for its support in the present work.

7. References

[1] S. F. Ali, M. R. Abdulrazzaq, and M. T. Gaata, (2025), "Learning Techniques-Based Malware Detection: A Comprehensive Review," *Mesopotamian Journal of CyberSecurity*, vol. 5, no. 1, pp. 273–300.

[2] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu, (2020), "A Review of Android Malware Detection Approaches Based on Machine Learning," *IEEE Access*, vol. 8, pp. 124579–124607, doi: 10.1109/ACCESS.2020.3006143.

[3] StatCounter 1999-2024, (2026), "Mobile Operating System Market Share Iraq," https://gs.statcounter.com/os-market-share/mobile/iraq, [Online].

[4] Y. Pan, X. Ge, C. Fang, and Y. Fan, (2020), "A Systematic Literature Review of Android Malware Detection Using Static Analysis," *IEEE Access*, vol. 8, pp. 116363–116379, doi:10.1109/ACCESS.2020.3002842.

[5] S. K. Smmarwar, G. P. Gupta, and S. Kumar, (2024), "Android malware detection and identification frameworks by leveraging the machine and deep learning techniques: A comprehensive review," *Elsevier B.V.* doi: 10.1016/j.teler.2024.100130.

[6] F. Alswaina and K. Elleithy, (2020), "Android malware family classification and analysis: Current status and future directions," *MDPI AG*. doi: 10.3390/electronics9060942.

[7] R. A. Yunmar, S. S. Kusumawardani, Widyawan, and F. Mohsen, (2024), "Hybrid Android Malware Detection: A Review of Heuristic-Based Approach," *IEEE Access*, vol. 12, pp. 41255–41286, doi:10.1109/ACCESS.2024.3377658.

[8] A. Alzubaidi, (2021), "Recent Advances in Android Mobile Malware Detection: A Systematic Literature Review," *Institute of Electrical and Electronics Engineers Inc.* doi:10.1109/ACCESS.2021.3123187.

[9] S. R. T. Mat, M. F. A. Razak, M. N. M. Kahar, J. M. Arif, and A. Firdaus, (2022), "A Bayesian probability model for Android malware detection," *ICT Express*, vol. 8, no. 3, pp. 424 – 431, doi:10.1016/j.icte.2021.09.003.

[10] Y. Yao *et al.*, (2024), "Research on Malware Detection Technology for Mobile Terminals Based on API Call Sequence," *Mathematics*, vol. 12, no. 1, doi: 10.3390/math12010020.

[11] O. E. Saied and K. H. Thanoon, (2025), "Static Analysis-based Detection of Android Malware using Machine Learning Algorithms," *Sistemasi: Jurnal Sistem Informasi*, vol. 14, no. 5, pp. 2540–2551.

[12] A. Pathak, T. S. Kumar, and U. Barman, (2024), "Static analysis framework for permission-based dataset generation and android malware detection using machine learning," *EURASIP J. Inf. Secur.*, vol. 2024, no. 1, p. 33.

[13] H. Kato, T. Sasaki, and I. Sasase, (2021), "Android Malware Detection Based on Composition Ratio of Permission Pairs," *IEEE Access*, vol. 9, pp. 130006–130019, doi:10.1109/ACCESS.2021.3113711.

[14] Z. Namrud, S. Kpodjedo, A. Bali, and C. Talhi, (2022), "Deep-Layer Clustering to Identify Permission Usage Patterns of Android App Categories," *IEEE Access*, vol. 10, pp. 24240–24254, doi:10.1109/ACCESS.2022.3156083.

[15] Y. Song, Y. Geng, J. Wang, S. Gao, and W. Shi, (2021), "Permission Sensitivity-Based Malicious Application Detection for Android," *Security and Communication Networks*, vol. (2021), doi:10.1155/2021/6689486.

[16] B. Mishra *et al.*, (2022), "Privacy Protection Framework for Android," *IEEE Access*, vol. 10, pp. 7973–7988, doi: 10.1109/ACCESS.2022.3142345.

[17] A. Banik and J. P. Singh, (2023), "Android Malware Detection by Correlated Real Permission Couples Using FP Growth Algorithm and Neural Networks," *IEEE Access*, vol. 11, pp. 124996–125010, doi: 10.1109/ACCESS.2023.3323845.

[18] D. Ö. Şahın, S. Akleylek, and E. Kiliç, (2022), "LinRegDroid: Detection of Android Malware Using Multiple Linear Regression Models-Based Classifiers," *IEEE Access*, vol. 10, pp. 14246–14259, doi: 10.1109/ACCESS.2022.3146363.

[19] R. Y. Mawoh, J. B. A. Wacka, F. Tchakounté, C. Fachkha, and Kolyang, (2025), "An accurate approach to discriminate android colluded malware from single app malware using permissions intelligence," *Sci. Rep.*, vol. 15, no. 1, doi: https://doi.org/10.1038/s41598-025-86568-w.

[20] M. A. Haq and M. Khuthaylah, (2024), "Leveraging Machine Learning for Android Malware Analysis: Insights from Static and Dynamic Techniques," *Engineering, Technology & Applied Science Research*, vol. 14, no. 4, pp. 15027–15032.

[21] M. İbrahim, B. Issa, and M. B. Jasser, (2022), "A Method for Automatic Android Malware Detection Based on Static Analysis and Deep Learning," *IEEE Access*, vol. 10, pp. 117334–117352, doi:10.1109/ACCESS.2022.3219047.

[22] J. Xu, Y. Li, R. H. Deng, and K. Xu, (2022), "SDAC: A Slow-Aging Solution for Android Malware Detection Using Semantic Distance Based API Clustering," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 2, pp. 1149–1163, doi:10.1109/TDSC.2020.3005088.

[23] Y.-C. Chen, H.-Y. Chen, T. Takahashi, B. Sun, and T.-N. Lin, (2021), "Impact of Code Deobfuscation and Feature Interaction in Android Malware Detection," *IEEE Access*, vol. 9, pp. 123208–123219, doi: 10.1109/ACCESS.2021.3110408.

[24] H. Alamro, W. Mtouaa, S. Aljameel, A. S. Salama, M. A. Hamza, and A. Y. Othman, (2023), "Automated Android Malware Detection Using Optimal Ensemble Learning Approach for Cybersecurity," *IEEE Access*, vol. 11, pp. 72509–72517, doi:10.1109/ACCESS.2023.3294263.

[25] F. Zhou, D. Wang, Y. Xiong, K. Sun, and W. Wang, (2025), "Androfim: few-shot android malware family detection based on image representation: F. Zhou *et al.*," *Cybersecurity*, vol. 8, no. 1, p. 69.

[26] O. E. Kural, E. Kiliç, and C. Aksaç, (2023), "Apk2Audio4AndMal: Audio Based Malware Family Detection Framework," *IEEE Access*, vol. 11, pp. 27527–27535, doi:10.1109/ACCESS.2023.3258377.

[27] R. Korine and D. Hendler, (2021), "DAEMON: Dataset/Platform-Agnostic Explainable Malware Classification Using Multi-Stage Feature Mining," *IEEE Access*, vol. 9, pp. 78382–78399, doi:10.1109/ACCESS.2021.3082173.

[28] F. Mercaldo and A. Santone, (2022), "Formal Equivalence Checking for Mobile Malware Detection and Family Classification," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2643–2657, doi: 10.1109/TSE.2021.3067061.

[29] D. Liang, L. Shen, Z. Chen, C. Ma, and J. Feng, (2022), "A Formal Method for Description and Decision of Android Apps Behavior Based on Process Algebra," *IEEE Access*, vol. 10, pp. 108668–108683, doi: 10.1109/ACCESS.2022.3210386.

[30] C. Chimeleze *et al.*, (2022), "BFEDroid: A Feature Selection Technique to Detect Malware in Android Apps Using Machine Learning," *Security and Communication Networks*, vol. (2022), doi:10.1155/2022/5339926.

[31] H. Zhu, Y. Li, R. Li, J. Li, Z. You, and H. Song, (2021), "SEDMDroid: An Enhanced Stacking Ensemble Framework for Android Malware Detection," *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 2, pp. 984–994, doi: 10.1109/TNSE.2020.2996379.

[32] C. Cilleruelo, Enrique-Larriba, L. De-Marcos, and J.-J. Martinez-Herráiz, (2021), "Malware Detection Inside App Stores Based on Lifespan Measurements," *IEEE Access*, vol. 9, pp.119967–119976, doi: 10.1109/ACCESS.2021.3107903.

[33] S. Garg and N. Baliyan, (2022), "MalVulDroid: Tracing Vulnerabilities from Malware in Android Using Natural Language Processing," *Journal of Web Engineering*, vol. 21, no. 8, pp. 2339–2361, doi: 10.13052/jwe1540-9589.2185.

[34] T. Kacem and S. Tossou, (2025), "Trandroid: An Android Mobile Threat Detection System Using Transformer Neural Networks," *Electronics (Basel).*, vol. 14, no. 6, p. 1230, doi:https://doi.org/10.3390/electronics14061230.

[35] H. J. Hadi, A. Khalid, F. B. Hussain, N. Ahmad, and M. A. Alshara, (2025), "FLSH: A Framework Leveraging Similarity Hashing for Android Malware and Variant Detection," *IEEE Access*, vol. 13, pp. 26142–26156, doi: https://doi.org/10.1109/access.2025.3537110.

[36] H.-J. Zhu, L.-M. Wang, S. Zhong, Y. Li, and V. S. Sheng, (2022), "A Hybrid Deep Network Framework for Android Malware Detection," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 12, pp. 5558–5570, doi: 10.1109/TKDE.2021.3067658.

[37] W. Yuan, Y. Jiang, H. Li, and M. Cai, (2021), "A Lightweight On-Device Detection Method for Android Malware," *IEEE Trans. Syst. Man Cybern. Syst.*, vol. 51, no. 9, pp. 5600–5611, doi:10.1109/TSMC.2019.2958382.

[38] B. B. Gupta *et al.*, (2025), "Earthworm Optimization Algorithm Based Cascade LSTM-GRU Model for Android Malware Detection," *Cyber Security and Applications*, p. 100083, doi:https://doi.org/10.1016/j.csa.2024.100083.

[39] M. Rashid *et al.*, (2025), "Hybrid Android Malware Detection and Classification Using Deep Neural Networks," *International Journal of Computational Intelligence Systems*, vol. 18, no. 1, doi:https://doi.org/10.1007/s44196-025-00783-x.

[40] H. Chen, Z. Li, Q. Jiang, A. Rasool, and L. Chen, (2021), "A hierarchical approach for android malware detection using authorization-sensitive features," *Electronics (Switzerland)*, vol. 10, no. 4, pp. 1 – 24, doi: 10.3390/electronics10040432.

[41] E. Odat and Q. M. Yaseen, (2023), "A Novel Machine Learning Approach for Android Malware Detection Based on the Co-Existence of Features," *IEEE Access*, vol. 11, pp. 15471–15484, doi: 10.1109/ACCESS.2023.3244656.

[42] Z. Meng *et al.*, (2025), "Detecting Android Malware by Visualizing App Behaviors from Multiple Complementary Views," *IEEE Transactions on Information Forensics and Security*, p. 1, doi:https://doi.org/10.1109/tifs.2025.3547301.

[43] Z. Meng *et al.*, (2025), "MalFlows: Context-aware Fusion of Heterogeneous Flow Semantics for Android Malware Detection," *arXiv preprint arXiv:2508.03588*.

[44] X. Xu, S. Jiang, J. Zhao, and X. Wang, (2024), "DCEL: Classifier Fusion Model for Android Malware Detection," *Journal of Systems Engineering and Electronics*, vol. 35, no. 1, pp. 163–177, doi:10.23919/JSEE.2024.000018.

[45] S. Nethala, P. Chopra, K. Kamaluddin, S. Alam, S. A. Alharbi, and M. Alsaffar, (2025), "A Deep Learning-Based Ensemble Framework for Robust Android Malware Detection," *IEEE Access*, p. 1, doi: https://doi.org/10.1109/access.2025.3551152.

[46] Ö. Kiraz and Ibrahim Alper Doğru, (2024), "Visualising static features and classifying android malware using a convolutional neural network approach," *Applied Sciences*, vol. 14, no. 11, p.4772.

[47] A. Al Tawil, D. Qawasmeh, and B. Qawasmeh, (2025), "Enhanced Parrot Optimizer Algorithm: A Proposed Method for Optimized Malware Classification," *International Journal of Artificial Intelligence Applications*, vol. 1, no. 1.

[48] A. O. Otiko and G. A. Iyang, (2025), "Android Malware Classification with Feature Selection using Artificial Bee Colony Algorithm," *Available at SSRN 5293728*.

[49] I. Almomani *et al.*, (2021), "Android Ransomware Detection Based on a Hybrid Evolutionary Approach in the Context of Highly Imbalanced Data," *IEEE Access*, vol. 9, pp. 57674–57691, doi:10.1109/ACCESS.2021.3071450.

[50] A. Taha, A. H. Osman, and Y. S. Baguda, (2025), "An Intelligent Technique for Android Malware Identification Using Fuzzy Rank-Based Fusion," *Technologies (Basel).*, vol. 13, no. 2, p. 45, doi: https://doi.org/10.3390/technologies13020045.

[51] R. S. Arslan, (2025), "JDroid: Android malware detection using hybrid opcode feature vector," *PeerJ Comput. Sci.*, vol. 11, p. e3051.

[52] B. Urooj, M. A. Shah, C. Maple, M. K. Abbasi, and S. Riasat, (2022), "Malware Detection: A Framework for Reverse Engineered Android Applications Through Machine Learning Algorithms," *IEEE Access*, vol. 10, pp. 89031–89050, doi:10.1109/ACCESS.2022.3149053.

[53] F. A. Almarshad, M. Zakariah, G. A. Gashgari, E. A. Aldakheel, and A. I. A. Alzahrani, (2023), "Detection of Android Malware Using Machine Learning and Siamese Shot Learning Technique for Security," *IEEE Access*, vol. 11, pp. 127697–127714, doi:10.1109/ACCESS.2023.3331739.

[54] L. da Costa and V. Moia, (2023), "A Lightweight and Multi-Stage Approach for Android Malware Detection Using Non-Invasive Machine Learning Techniques," *IEEE Access*, vol. 11, pp. 73127–73144, doi: 10.1109/ACCESS.2023.3296606.

[55] M. M. Alani and A. I. Awad, (2022), "PAIRED: An Explainable Lightweight Android Malware Detection System," *IEEE Access*, vol. 10, pp. 73214–73228, doi:10.1109/ACCESS.2022.3189645.

[56] P. R. Sekhar, V. Sohana, R. Shaik, P. Thanuja, and K. J. Mary, (2025), "Lightweight Machine Learning-Based Static Analysis Framework for Real-Time Android Malware Detection," *Synthesis: A Multidisciplinary Research Journal*, vol. 3, no. 1s, pp. 60–69.

[57] J.-H. Shin, D. Y. Kim, and K. Lee, (2025), "Advanced Financial Fraud Malware Detection Method in the Android Environment," *Applied Sciences*, vol. 15, no. 7, p. 3905, doi:https://doi.org/10.3390/app15073905.

[58] M. Jalandra, M. Kuliha, and J. Kaur, (2025), "Hybrid Machine Learning Model for Android Ransomware Detection Using URL-Based Features," *IJSAT-International Journal on Science and Technology*, vol. 16, no. 3.

[59] M. Tanvirul Alam, D. Bhusal, and N. Rastogi, (2024), "Revisiting Static Feature-Based Android Malware Detection," *arXiv e-prints*, p. arXiv–2409.

[60] A. Faiz, M. Fuzail, A. Aftab, N. Aslam, and A. J. Akbar, (2024), "A Multimodal Machine Learning Approach for Android Malware Detection: Static Code Analysis," *International Journal of Information Systems and Computer Technologies*, vol. 4, no. 1, pp. 20–35.

[61] S. Akbar and T. Khan, (2025), "Malware Detection Using Static Feature Analysis and Deep Learning Techniques: A Static Robust DL-Based Model for Android Malware Detection," *Information Technology and Control*, vol. 54, no. 4, pp. 1358–1382.

[62] H. Berger, C. Hajaj, E. Mariconti, and A. Dvir, (2022), "Crystal Ball: From Innovative Attacks to Attack Effectiveness Classifier," *IEEE Access*, vol. 10, pp. 1317–1333, doi:10.1109/ACCESS.2021.3138628.

[63] G. Xu *et al.*, (2024), "OFEI: A Semi-Black-Box Android Adversarial Sample Attack Framework Against DLaaS," *IEEE Transactions on Computers*, vol. 73, no. 4, pp. 956–969, Apr. doi:10.1109/TC.2023.3236872.

[64] G. You, G. Kim, S. Han, M. Park, and S.-J. Cho, (2022), "Deoptfuscator: Defeating Advanced Control-Flow Obfuscation Using Android Runtime (ART)," *IEEE Access*, vol. 10, pp. 61426–61440, doi: 10.1109/ACCESS.2022.3181373.

[65] D. Li, S. Cui, Y. Li, J. Xu, F. Xiao, and S. Xu, (2024), "PAD: Towards Principled Adversarial Malware Detection Against Evasion Attacks," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 2, pp. 920–936, doi: 10.1109/TDSC.2023.3265665.

[66] A. Rashid and J. Such, (2023), "MalProtect: Stateful Defense Against Adversarial Query Attacks in ML-Based Malware Detection," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 4361–4376, doi: 10.1109/TIFS.2023.3293959.

[67] R. Yumlembam, B. Issac, S. M. Jacob, and L. Yang, (2023), "IoT-Based Android Malware Detection Using Graph Neural Network With Adversarial Defense," *IEEE Internet Things J.*, vol. 10, no. 10, pp. 8432–8444, doi: 10.1109/JIOT.2022.3188583.

[68] J. Wang, X. H. Liu, M. Huang, P. Zhou, and Y. Xu, (2025), "Android Malware Detection Method Combining Multi-Frequency Features and Convolutional Neural Networks," *IEEE Access*, p. 1, doi: https://doi.org/10.1109/access.2025.3550124.

[69] L. N. Vu and S. Jung, (2021), "AdMat: A CNN-on-Matrix Approach to Android Malware Detection and Classification," *IEEE Access*, vol. 9, pp. 39680–39694, doi:10.1109/ACCESS.2021.3063748.

[70] Y. Hei *et al.*, (2024), "Hawk: Rapid Android Malware Detection Through Heterogeneous Graph Attention Networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 35, no. 4, pp. 4703–4717, doi: 10.1109/TNNLS.2021.3105617.

[71] Y. Liu, M. G. M. Johar, and J. Tham, (2025), "An Android Malware Detection Method Based on MLSTM," *Journal of Digital Information Management*, vol. 23, no. 1, pp. 11–25, doi:https://doi.org/10.6025/jdim/2025/23/1/11-25.

[72] H. Zhao, Y. Wu, D. Zou, and H. Jin, (2024), "An Empirical Study on Android Malware Characterization by Social Network Analysis," *IEEE Trans. Reliab.*, vol. 73, no. 1, pp. 757–770, doi:10.1109/TR.2023.3304389.

[73] I. Almomani, A. Alkhayer, and W. El-Shafai, (2022), "An Automated Vision-Based Deep Learning Model for Efficient Detection of Android Malware Attacks," *IEEE Access*, vol. 10, pp. 2700–2720, doi: 10.1109/ACCESS.2022.3140341.

[74] J. Yang, Y. Li, H. Ren, D. Jia, and X. Wang, (2025), "SAC: Collaborative learning of structure and content features for Android malware detection framework," *Neurocomputing*, p. 130053, doi:https://doi.org/10.1016/j.neucom.2025.130053.

[75] H. Rodriguez-Bazan, G. Sidorov, and P. J. Escamilla-Ambrosio, (2023), "Android Ransomware Analysis Using Convolutional Neural Network and Fuzzy Hashing Features," *IEEE Access*, vol. 11, pp. 121724–121738, doi: 10.1109/ACCESS.2023.3328314.

[76] B. Menaouer, A. El Hadj Mohamed Islem, and M. Nada, (2023), "Android Malware Detection Approach Using Stacked AutoEncoder and Convolutional Neural Networks," *International Journal of Intelligent Information Technologies*, vol. 19, no. 1, doi:10.4018/IJIIT.329956.

[77] J. Xu and Q. Yuan, (2022), "LibRoad: Rapid, Online, and Accurate Detection of TPLs on Android," *IEEE Trans. Mob. Comput.*, vol. 21, no. 1, pp. 167–180, doi:10.1109/TMC.2020.3003336.

[78] J. Qiu, X. Yang, H. Wu, Y. Zhou, J. Li, and J. Ma, (2022), "LibCapsule: Complete Confinement of Third-Party Libraries in Android Applications," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 5, pp. 2873–2889, doi: 10.1109/TDSC.2021.3075817.

[79] C. Sun, Y. Ma, D. Zeng, G. Tan, S. Ma, and Y. Wu, (2023), "μDep: Mutation-Based Dependency Generation for Precise Taint Analysis on Android Native Code," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 2, pp. 1461–1475, doi: 10.1109/TDSC.2022.3155693.

[80] C. Bai, Q. Han, G. Mezzour, F. Pierazzi, and V. S. Subrahmanian, (2021), "DBankDBank: Predictive Behavioral Analysis of Recent Android Banking Trojans," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 3, pp. 1378–1393, doi: 10.1109/TDSC.2019.2909902.

[81] Y. Cui, Y. Sun, J. Luo, Y. Huang, Y. Zhou, and X. Li, (2022), "MMPD: A Novel Malicious PDF File Detector for Mobile Robots," *IEEE Sens. J.*, vol. 22, no. 18, pp. 17583–17592, doi:10.1109/JSEN.2020.3029083.

[82] M. Dener, G. Ok, and A. Orman, (2022), "Malware Detection Using Memory Analysis Data in Big Data Environment," *Applied Sciences (Switzerland)*, vol. 12, no. 17, doi:10.3390/app12178604.

[83] S. Seneviratne, R. Shariffdeen, S. Rasnayaka, and N. Kasthuriarachchi, (2022), "Self-Supervised Vision Transformers for Malware Detection," *IEEE Access*, vol. 10, pp. 103121–103135, doi:10.1109/ACCESS.2022.3206445.

[84] M. Naseer *et al.*, (2025), "Obfuscated Malware Detection and Classification in Network Traffic Leveraging Hybrid Large Language Models and Synthetic Data," *Sensors*, vol. 25, no. 1, p. 202, doi: https://doi.org/10.3390/s25010202.

[85] M. Maray, M. Maashi, H. M. Alshahrani, S. S. Aljameel, S. Abdelbagi, and A. S. Salama, (2024), "Intelligent Pattern Recognition Using Equilibrium Optimizer With Deep Learning Model for Android Malware Detection," *IEEE Access*, vol. 12, pp. 24516–24524, doi:10.1109/ACCESS.2024.3357944.

[86] J. Yu, Y. He, Q. Yan, and X. Kang, (2021), "SpecView: Malware Spectrum Visualization Framework With Singular Spectrum Transformation," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 5093–5107, doi: 10.1109/TIFS.2021.3124725.