Unraveling the Key of Machine Learning-based Android Malware Detection

JIAHAO LIU, National University of Singapore, Singapore
JUN ZENG*, National University of Singapore, Singapore
FABIO PIERAZZI, University College London, United Kingdom
ZIQI YANG, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, China
LORENZO CAVALLARO, University College London, United Kingdom
ZHENKAI LIANG, National University of Singapore, Singapore

With the rapid advancement of machine learning (ML), ML-based Android malware detection has gained significant popularity due to its ability to automatically learn malicious patterns from Android apps. However, the lack of an in-depth and systematic analysis of existing research makes it difficult to obtain a holistic understanding of the state of the art in this field. In this work, we present the most comprehensive investigation to date of ML-based Android malware detection systems, combining both empirical and quantitative analyses. We first organize prior work into a unified taxonomy based on Android app representations and the ML modeling pipeline. Building on this taxonomy, we design a general-purpose framework for ML-based Android malware detection and re-implement 12 representative approaches from three research communities—software engineering, security, and machine learning. Using this framework, we conduct a large-scale evaluation across three key dimensions: detection effectiveness, robustness to real-world challenges, and efficiency. Despite extensive research efforts and encouraging results, our findings reveal that existing learning-based Android malware detectors still face significant challenges, including vulnerability to malware evolution and susceptibility to adversarial attacks. We attribute these limitations to the detectors' ability to capture and leverage malware semantics, defined as semantic information that characterizes malicious behaviors derived from APK features. Finally, we summarize our key insights and provide actionable recommendations to guide future research in this domain.

CCS Concepts: • **Software and its engineering**; • **Security and privacy**;

Additional Key Words and Phrases: Android Malware Detection, Machine Learning, Systematization of Knowledge

1 Introduction

Over the past decade, ML-based Android malware detection has attracted increasing attention from various research communities, such as software engineering, security, and machine learning [12, 17, 48, 64, 67, 79, 96, 97,

*For correspondence, please contact Jiahao Liu and Jun Zeng.

Authors' Contact Information: Jiahao Liu, National University of Singapore, Singapore, Singapore, ljiahaomail@gmail.com, jiahao99@comp.nus.edu.sg; Jun Zeng, National University of Singapore, Singapore, Singapore, junzeng@comp.nus.edu.sg; Fabio Pierazzi, University College London, London, United Kingdom, f.pierazzi@ucl.ac.uk; Ziqi Yang, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou, China, yangziqi@zju.edu.cn; Lorenzo Cavallaro, University College London, London, United Kingdom, l.cavallaro@ucl.ac.uk; Zhenkai Liang, National University of Singapore, Singapore, Singapore, liangzk@comp.nus.edu.sg.

ACM 1557-7392/2026/4-ART
https://doi.org/10.1145/3809491

102]. Much attention has been given to exploring various combinations of APK features and ML models. The trend is primarily led by advancements in ML models (*e.g.*, graph neural network [48]) and analogies drawn from other well-studied fields (*e.g.*, social network [96]). Most existing approaches report high F1 scores (up to 0.98) [56, 96]. Such promising results motivate us to ask a number of important research questions regarding the current state of ML-based Android malware detection. For example, How do existing approaches represent and incorporate diverse APK features into ML models? How do different detection methods compare when evaluated under the same datasets, metrics, and toolchains? Are current ML-based approaches sufficient to meet real-world deployment requirements? Does employing more powerful ML models necessarily lead to better detection performance? Does model selection materially affect detection effectiveness when the feature set is fixed? Does incorporating more features to describe app behavior always improve performance? Finally, is there a positive correlation between detection effectiveness and computational efficiency?

Existing studies [35, 38, 64, 79] provide a good starting point to understand the landscape of ML-based Android malware detection. However, most are either theoretical reviews or limited in experimental scope and scalability, and thus fall short of a systematic, end-to-end view of the field. For example, Qiu et al. [79] organize the landscape by model family (*e.g.,* CNN and RNN) and examine how these models are applied in Android malware detection. To our knowledge, no prior work provides an empirical and quantitative real-world analysis that jointly considers effectiveness, robustness, and efficiency. Additionally, there is no publicly available framework that standardizes the implementation and evaluation of Android malware detectors, enabling reproducible research and supporting future development.

Accordingly, we conduct an empirical, quantitative study to answer these questions and clarify the current state of ML-based Android malware detection. Given the scale of the ecosystem — Google Play's catalog reached 4.2 million apps in 2025, a 6.3% increase over 2024 [5] — we prioritize methods that scale to large datasets and are practical for real-world deployment. Specifically, we implement and evaluate representative approaches to characterize both progress and remaining gaps. While informative, this exercise also reveals several challenges to fair comparisons as follows.

- *Unfair Comparisons.* Previous approaches are usually evaluated on datasets of different sizes, goodware-to-malware ratios, and training-to-testing ratios [17, 50, 61]. Moreover, they report outcomes using diverse metrics (*e.g.,* F1-score, Accuracy, and False Positive Rate), making it unclear which method performs better under specific settings. In addition, they often rely on different toolchains (*e.g.,* Androguard [1] and APKTool [3]) to develop detectors, which introduce ambiguity — whether the promising results stem from the novelty of the approach or not [89].
- *Unrealistic Evaluations.* Android malware detectors face a rapid threat landscape [21], with malware continually evolving to evade detection. The growing usage of obfuscation also makes malware harder to detect as its malicious intentions are hidden [15]. Additionally, ML models can be tricked by intentionally perturbed inputs [24, 44, 78, 88]. Although the impacts of some of these scenarios have been studied [21, 25, 53, 77], the lack of a comprehensive assessment creates a gap in our understanding of the main challenges that hinder the adoption of Android malware detection in real-world scenarios.
- *Unclear Computational Costs.* With the exponential growth of apps in sizes and complexities, feature extraction gradually becomes notably time-consuming — for example, it can take up 30 minutes to gather API paths from one app [99]. Furthermore, as ML models evolve in complexity, they demand greater computational power to achieve state-of-the-art results. Unfortunately, it remains uncertain what prices to pay to gain the desired results.

To address these challenges, we design FrameDroid, a general-purpose framework that streamlines the implementation and evaluation of ML-based Android malware detection systems. FrameDroid adopts a modular and configurable architecture, enabling both the rapid development of new detection techniques and the flexible

construction of diverse evaluation scenarios. Specifically, we begin with decomposing the detection pipeline into three core phases: APK characterization, feature representation, and ML modeling. Guided by this taxonomy, we conduct an empirical analysis of prior work to examine how existing approaches detect Android malware.

Next, we apply our framework to 12 representative approaches spanning three research communities: software engineering [94, 96, 97], security [17, 48, 56, 65, 67, 99, 102], and machine learning [50, 61]. To ensure fairness and reproducibility, we standardize shared tasks (*e.g.,* feature extraction) across different methods by using the same toolchain (*i.e.,* Androguard [1]), eliminating discrepancies caused by heterogeneous tool support. Furthermore, to facilitate the development of new detectors, FrameDroid is implemented as a modular and configurable system, allowing components (*e.g.,* neural networks) to be easily replaced or customized.

For a comprehensive assessment, we randomly sample 221,310 apps spanning ten years (2011-2020) from AndroZoo [13], following prior studies [28, 77]. AndroZoo is a continuously updated public repository of Android applications, and this dataset serves as our primary benchmark. We set the malware ratio to 10% to reflect realistic deployment conditions [77]. To systematically investigate the state of ML-based Android malware detection, we evaluate the selected approaches using commonly adopted metrics (*e.g.*, F1-score and accuracy), focusing on multiple dimensions: detection effectiveness and efficiency, robustness to app evolution and obfuscation, and resilience against adversarial attacks. To further validate the generalizability of our findings, we construct an additional dataset consisting of 7,911 apps collected between 2021 and 2024, including 7,120 benign apps from AndroZoo and 791 malicious apps from VirusTotal and AndroZoo. We evaluate the effectiveness of the selected approaches on this supplementary dataset to examine whether our key observations remain valid on more recent data.

Findings and Recommendations. Through our empirical and quantitative analysis, we present a holistic view of the state of the art in ML-based Android malware detection. Our results show that APK characterization and ML modeling remain central themes in the literature and continue to serve as the foundation for building effective detectors. Under identical experimental settings, many recent approaches achieve comparable performance. However, their effectiveness still degrades in challenging scenarios, such as those involving limited training data or adversarial attacks. Our analysis indicates that the key factor underlying detection effectiveness is the ability to extract semantic information that characterizes malicious behaviors from APK features, which we refer to as malware semantics. From the feature-design perspective, we observe that naively incorporating additional features does not necessarily improve performance; instead, irrelevant or weakly related features may dilute meaningful semantic signals and even harm detection accuracy. From the modeling perspective, we find that employing more powerful ML models alone does not guarantee better detection outcomes, particularly when the input features fail to adequately capture app behavior. Furthermore, given specific feature sets (e.g., API calls and permissions), ensemble-based models and deep learning models consistently outperform some traditional classifiers, suggesting that these models constitute strong and practical baselines for future detector design. We also observe that increased computational overhead is not a reliable indicator of improved detection capability, highlighting the need to carefully balance model complexity and performance. Overall, our findings suggest that future research should place greater emphasis on designing robust and practical detectors for real-world deployment, with careful consideration of both effectiveness and efficiency. Additional discussions and insights are provided in Section 6.

We emphasize that the design of effective ML-based Android malware detection solutions should be driven by the extraction and integration of malware semantics — that is, semantic representations of malicious app behaviors derived from APK features. Defining, modeling, and quantifying such semantics remains an open research challenge, yet it is critical for advancing the effectiveness and robustness of ML-based detection methods. We hope that our findings help inform and guide future research in this area, including — but not limited to — feature selection, model design, and the efficient allocation of computational resources.

In summary, we make the following contributions:

- We conduct a thorough systematic investigation of ML-based Android malware detection using empirical and qualitative methods, drawing a holistic picture of the field.
- We design a general-purpose framework, FrameDroid, to facilitate the implementation and evaluation of various Android malware detectors. For comparison in realistic settings, we collect the largest dataset to date, both in size and temporal coverage. To promote reproducibility and future research, we release our framework and dataset at https://github.com/ljiahao/FrameDroid.
- We offer a comprehensive comparative analysis of 12 representative approaches using FrameDroid, focusing on assessing their effectiveness, robustness, and efficiency. We point out that the key to enhancing ML-based Android malware detection is incorporating the malware semantics derived from APK features.

2 Learning-based Android Malware Detection

Android malware detection involves two steps: characterizing APKs and identifying malicious patterns. Recent trends have seen a shift towards using static feature extraction for APK profiling due to its efficiency and scalability [64, 79]. For malicious pattern identification, ML models have become increasingly popular since they can automatically learn patterns from features [61, 67, 96]. To validate the trend towards ML-based Android malware detection, we conduct a systematic investigation of the literature, which can be found in Section 7.

It is worth noting that, in an effort to better characterize the landscape of ML-based Android malware detection, a number of studies [35, 38, 64, 79, 80] have reviewed existing detection approaches to identify common practices and open challenges. However, these reviews largely rely on theoretical analyses without experimental validation or adopt narrow perspectives — for example, focusing primarily on how popular ML models are applied to this domain [79] or examining individual solutions in isolation. Such limitations make it difficult to draw broader conclusions or form a holistic understanding of how the entire detection pipeline operates, including what features are commonly used and how these features are utilized by ML models. Gao et al. [38] attempt to experimentally assess the state of ML-based Android malware detection; however, the analysis remains limited in scope and scale (*e.g.,* focusing mainly on robustness) and does not account for real-world settings or end-to-end performance evaluation. To the best of our knowledge, no existing work provides a comprehensive study that systematically investigates ML-based Android malware detection through both (i) empirical categorization and elucidate the overall detection workflow, and (ii) quantitative evaluation to assess the state of the art across multiple dimensions — effectiveness, robustness, and efficiency — under realistic conditions. Moreover, there is currently no publicly available, general-purpose framework that supports the development and evaluation of ML-based Android malware detection systems.

To bridge this gap, we first present a systematic investigation of ML-based Android malware detection. We then design and release a general-purpose framework, FrameDroid, together with a large-scale, realistic dataset (see Section 3), enabling the most comprehensive and wide-ranging comparative analysis to date. Using this unified platform, we examine the current state of ML-based Android malware detection and highlight the key challenges and opportunities that shape this field.

In this section, we demonstrate the empirical analysis about how existing detectors represent and incorporate features for Android malware detection. Rather than analyzing each approach individually, our methodology is to deconstruct and unify the workflow of ML-based Android malware detection. We identify three common phases: APK characterization (Section 2.1), feature representation (Section 2.2), and ML modeling (Section 2.3). Following this, we collate and summarize the key techniques employed in each phase by investigating existing approaches, offering a thorough overview of how ML models are applied in Android malware detection.

Manifest	$\mathbb{M}$	$\Rightarrow$	$Hardware\ Component \mid Intent \mid$ $Application\ Component \mid Permission$
DEX	$\mathbb{D}$	$\Rightarrow$	$ByteCode \mid Opcode \mid Intent \mid Code\ String \mid$ $API\ Call\ Information(API\ Call/Program\ Graph)$
Library	$\mathbb{L}$	$\Rightarrow$	$ByteCode \mid Opcode$
Resource	$\mathbb{R}$	$\Rightarrow$	$Resource\ Information$

Fig. 1. APK files and their corresponding features.

2.1 APK Characterization

Input from APK. An APK is a compressed archive that contains an app's codebase, resources, and auxiliary files. It contains several types of files and directories, including the Manifest ($\mathbb{M}$), Dex ($\mathbb{D}$), Library ($\mathbb{L}$), and Resource ($\mathbb{R}$) components [35, 93], each contributing distinctive features relevant to malware detection. *Manifest*: The manifest serves as a descriptor file that provides essential metadata about the app, including the package name, requested permissions, components, and hardware requirements. *Dex*: This includes Java classes that are compiled according to the Dalvik Executable (DEX) file standard, designed to run on the Dalvik Virtual Machine (DVM). *Library*: Native libraries appear as shared object files and offer critical low-level functionality (*e.g.,* WebKit). They support and optimize the app's execution and may expose behaviors indicative of malicious activity. *Resource*: This category includes static assets and non-compiled resources required by the app, such as images, XML layouts, and animation sequences.

APK features. The quartet of file types described above forms the foundation of APK characterization. Researchers commonly employ static analysis techniques to extract and distill relevant features. *Manifest* and *Resource* files typically follow well-defined structures, such as XML, which makes them straightforward to parse. Features from these components can thus be efficiently extracted using regular expressions or dedicated XML parsers. In contrast, the *Dex* and *Library* components consist of binary files, making their analysis substantially more involved. Feature extraction from these binaries requires reverse engineering techniques and specialized tooling. *Dex* can be disassembled by Androguard [1] and APKTool [3] into smali code, which is a more human-readable representation depicting apps' behaviors. For the *Library*, tools like Angr [2] or IDA Pro [6] are helpful in disassembling the native libraries into assembly codes, which facilitates analysis of the native services utilized by apps. By analyzing this assembly code, it is possible to capture critical features within native code.

Figure 1 delineates the features typically utilized and their corresponding APK file or folder in ML-based Android malware detection. In the subsequent section, we elaborate on these features in detail.

[$\mathbb{M}$] *Hardware Component.* Android apps employ certain hardware components (*e.g.,* camera) to execute particular functions (*e.g.,* taking photos). The request for specific hardware components carries distinct security implications, as the utilization of hardware combinations often indicates potentially harmful behaviors [17]. For instance, an app utilizing the camera and network may have the capability to monitor user activities and transmit this data to remote servers. Recognizing the potential of this insight, several approaches [17, 56, 92, 102] have exploited these features as a heuristic for Android malware detection.

[$\mathbb{M}$] *Application Component.* An APK uses four primary components, namely, Activity, Service, Content Provider, and Broadcast Receiver, to provide different entry points for the system/users. Specifically, Activity provides interfaces for direct user engagement; Service sustains the app's background operations; Broadcast Receiver delivers system-wide events to the app, and Content Provider manages a shared set of app data. Commonly, one malware family employs similar component names, such as *SearchService* in the DroidKungFu family [7]. Inspired by this, application components are utilized to capture similar fingerprints in Android malware [56, 92, 102].

[𝕄|𝔻] *Intent.* As the primary ways of communication among components, intents connect various Application Components and delineate standard operations one app can perform. They are pivotal in initiating Activities, managing Services lifecycle, and delivering broadcast information to Broadcast Receivers. Malware often monitors these communication status to trigger malicious actions, such as activating pre-configured malicious activities [113]. Approaches [37, 61, 101] aim to capture these malicious behaviors by analyzing corresponding intents.

[𝕄] *Permission.* Android employs a permission-based mechanism to regulate access to sensitive data and restricted actions. For an app to carry out particular actions, it must obtain the requisite permissions [19, 20]. The set of permissions required by an app can thus offer insights into its intended behaviors. Particularly, malware often demands permissions that are unnecessary for benign apps to execute malicious actions [33]. Consequently, numerous Android malware detectors [17, 48, 56, 94, 102] capture the differences to distinguish malware.

[𝔻] *API Call Information (API Call and Program Graph).* API Call Information, consisting of API calls and their connections, is a widely utilized feature source in Android malware detection. Android apps make use of these API calls to access the operating system's functionality and system resources [74]. For instance, the invocation of *sendTextMessage()* suggests that the app is likely to send a text message. Furthermore, API calls are connected to form a graph, where each node signifies a method, and each edge denotes a method invocation [59], illustrating the app's structural information [96]. For clarity, we differentiate API Call Information into two sub-categories: *API Call*, referring to the individual API calls, and *Program Graph*, denoting the relationships among these API calls. Numerous studies [17, 56, 94] detect sensitive API calls (*e.g., getDeviceId()*) to estimate the probability of an app being malicious. In contrast, other solutions [48, 50, 66, 96, 99, 105] venture deeper, examining the relationships between API calls to capture an app's semantics for malware detection.

[𝔻|𝕃] *ByteCode and Opcode.* Similar to previous research [29, 62, 85, 102], we treat both the *raw bytecode* and the *assembly code* derived from the *Dex* and *Library* as ByteCode. ByteCode contains a sequence of instructions, where each instruction consists of a single Opcode and several operands. The Opcode denotes a specific operation; for instance, the `invoke` means a method invocation. The operands provide additional information for the Opcode, such as the method name. In addition, ByteCode and Opcode from the *Dex* and *Library* offer insights into the static execution flows of apps' Java and Native codes, providing a view of how an app work [56]. Recent research attempts to represent Bytecode and Opcode in various formats, such as image [14, 51, 67, 98] and text [54, 102, 103], to capture apps' semantics.

[𝔻] *Code String.* Apps often embed key information like URLs and IP addresses as string values within their codebase. These strings can be traced in the assembly code, tagged either as `const-string` or `const-string/jumbo`. Such strings can provide crucial clues about potential malicious activities [17]. For instance, malware sets up network sockets to communicate with remote servers, using the string `socket` in the codebase. A number of works [17, 56, 115] have used code strings to identify potential illegal operations.

[ℝ] *Resource Information.* Apps utilize resources to house traditional files and static elements, such as bitmaps and animation instructions. These resources are generally decoupled from the application codebase for ease of maintenance. Attackers sometimes embed malicious code within resource files, like image files, as a tactic to evade detection. Approaches like DeepRefiner [102] consider resource information as a crucial feature source for malware detection.

2.2 Feature Representation

APK characterizations provide multi-perspective views of an app's behavior. Before these features can be fed into ML models, they must be encoded into representations that the models can interpret. Broadly, the encoding process falls into four categories: categorical, image-based, text-based, and graph-based. Figure 2 illustrates the

$$
\begin{array}{lcl}
\text{Categorical} & \Rightarrow & \mathit{Hardware\ Component} \mid \mathit{API\ Call} \mid \mathit{ByteCode} \mid \\
 & & \mathit{Resource\ Information} \mid \mathit{Opcode} \mid \mathit{Code\ String} \mid \\
 & & \mathit{Application\ Component} \mid \mathit{Permission} \mid \mathit{Intent} \\
\text{Image-based} & \Rightarrow & \mathit{ByteCode} \mid \mathit{Opcode} \mid \mathit{API\ Call} \\
\text{Text-based} & \Rightarrow & \mathit{ByteCode} \mid \mathit{Opcode} \mid \mathit{API\ Call} \\
\text{Graph-based} & \Rightarrow & \mathit{Program\ Graph} \mid \mathit{API\ Call}
\end{array}
$$

Fig. 2. The relationships between Feature Representations and widely used APK Features.

relationships between these encoding strategies and the corresponding APK features. In the following section, we discuss each encoding strategy in detail.

Categorical encoding. Features like hardware components, intents, and code strings are often viewed as categorical data and can be easily transformed into numerical values. A widely-encoding strategy is to convert these features into a binary vector, where each position indicates whether a specific feature exists or not [17, 36, 50, 60, 61, 94, 106]. On the other hand, another line of research [56] calculates the frequency of each feature to obtain a vector of numerical values, reflecting the importance of each feature.

Image-based encoding. Representing specific features as images and leveraging image processing techniques is a well-established approach in malware detection. Specifically, bytecode and opcode are often visualized as images to describe apps' behaviors [29, 30, 51, 67, 98]. Notably, DexRay [29] transforms the app's bytecode into grey-scale vector images, wherein each pixel corresponds to a distinct byte. In a similar vein, some other features are also mapped to images to depict the APK's characteristics. For instance, Zegzhda et al. [107] combine API calls with protection levels as an RGB image.

Text-based encoding. Text-based encoding is also a widely used strategy in Android malware detection. Many existing methods [54, 55, 82, 99, 102] have approached APK features from a textual perspective, employing natural language processing (NLP) techniques to amplify detection capability. For example, by considering API calls as `words` and their sequences as `sentences`, methods presented in [54, 55, 99] utilize word embedding techniques, such as Word2Vec [69], to extract semantic information included in the API calls. Separately, Sun et al. [82] treat API calls and permissions as sparse data, applying Doc2Vec [58] to derive their vector representations.

Graph-based encoding. Recently, graph structure has been widely adopted to represent apps' semantics [48, 59]. One direction is to leverage program graphs to model APK behaviors [48, 75, 76, 96, 97]. Another avenue aims to build API-based feature graphs, drawing insights from API calls and their meta-relationships. An example is to identify whether two API calls are in the same block, thereby capturing the app's intended operations [50, 52]. When these features are represented as graphs, graph-based techniques (*e.g.,* Graph2Vec [71], social network [96]) are employed to extract apps' structural information.

2.3 Machine Learning Modeling

After encoding these features as numerical vectors, machine learning models are leveraged to identify malicious patterns. Following previous studies [79, 111], we categorize the models employed in Android malware detection into two main categories: traditional machine learning (TML) and deep learning (DL) models. TML models, such as linear regression or decision trees, typically exhibit simpler structures that can explicitly model the relationship between input and output. As such, these TML models often require domain knowledge to extract features from input data. In contrast, DL models are characterized by their multiple layers of neurons, enabling them to capture complex non-linear mappings from input to output [109]. This capability means that DL models are less reliant on domain knowledge during the feature extraction. In this section, we provide a concise introduction to the widely used models.

TML models. Given APK features, TML models are commonly employed to discern patterns from them. The Support Vector Machine (SVM) can find one hyperplane that separates the high-dimensional data points with varying labels. This capability has made it a popular choice to detect Android malware [17, 40, 50, 84, 99]. K-Nearest Neighbors (KNN) has also been applied in Android malware detection [12, 95–97]. This algorithm identifies the nearest neighbors of a given sample and subsequently classifies the sample based on the majority label of its neighbors. Additionally, as an ensemble-based learning method, Random Forest (RF) creates a forest of decision trees, each trained on a random subset of the data. This approach capitalizes on the strength of multiple decision trees, making the model more robust and accurate than individual trees. Such advantages have led to its significant application in malware detectors [65, 116].

DL models. DL models have exhibited strong capability in modeling malware behaviors. This part provides an insight into the widely used DL models tailored for Android malware detection. As a basic feed-forward neural network, Multi-Layer Perceptron (MLP), composed of several layers of neurons, has shown significant effectiveness in detecting Android malware [26, 56, 61, 66, 82, 114]. The Recurrent Neural Network (RNN) [68] is a type of neural architecture that can capture the sequential information of input data. By representing APK features (*e.g.*, API calls and bytecode) as sequences, existing solutions [90, 102, 103] leverage RNN to explore the temporal dependencies embedded in these features. The Convolutional Neural Network (CNN) [45] is equipped with multiple convolutional and pooling layers, enabling it to recognize contextual information derived from low-level features [41]. This intrinsic capability makes it especially effective in capturing patterns from image-oriented data. Reflecting its efficacy, CNN has been extensively employed to extract malicious patterns from image-based features in Android malware detection [36, 46, 51, 52, 55, 67]. Given the graph representation of APK features (*e.g.,* program graphs), Graph Neural Network (GNN) [57] can facilitate malware detection [34, 39, 48, 65]. This is because GNN can effectively propagate and aggregate node information along graph edges, thereby capturing the structural information of apps. Utilizing a process of encoding and subsequently decoding input features, Autoencoders (AE) have the capacity to generate refined data representations, which makes it a popular choice in detecting malware [61, 104, 117].

In addition, existing research [14, 81, 91] also tries to explore the potential of other DL models, like generative adversarial network (GAN) [42] and deep belief network (DBN) [49]. For instance, Amin et al. [14] employ the dual-network structure of GAN — one generates malware samples and the other works to distinguish these samples — to enhance the malware detection capability.

Finding: At the core of static and ML-based Android malware detection lies the ability to accurately *profile app behaviors* and to *select and leverage appropriate ML models* that can effectively uncover malicious patterns. Achieving this requires not only expressive feature representations but also a careful alignment between behavioral semantics and model capabilities.

3 FrameDroid

To support experimental reproducibility and fair comparison, we propose a general-purpose framework, FrameDroid, guided by the detection workflow discussed in Section 2, to facilitate the development and evaluation of new Android malware detectors. Figure 3 illustrates the overall architecture and workflow of FrameDroid. The process begins with a collection of apps, from which a set of features is extracted and stored in a *feature database*. These features are then processed by a *preprocessor*, which transforms them into numerical vectors. A selected *ML model* is subsequently trained and evaluated to detect malicious applications. Furthermore, FrameDroid provides configurable experimental settings to support diverse real-world scenarios, such as varying goodware-to-malware ratios, different training data sizes, and robustness assessments against adversarial attacks.

This modular design of FrameDroid simplifies the development of ML-based Android malware detectors. Three key modules, *feature database*, *preprocessor*, and *ML model*, captures the main aspects of designing ML

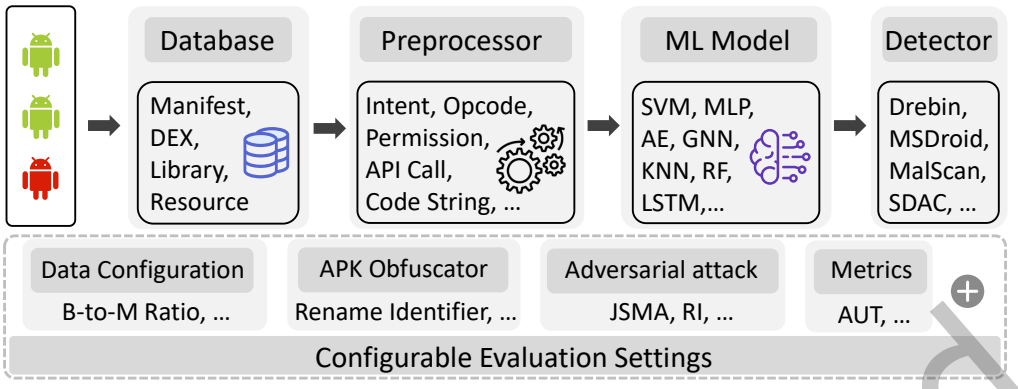


Fig. 3. The overall architecture of FrameDroid.

solutions for Android malware detection. The *feature database* organizes extracted features categorically according to their sources, as discussed in Section 2.1, including Manifest, Dex, Library, and Resource. For example, the Manifest category contains features extracted from the AndroidManifest.xml file, such as permissions, intents, and application components. Further details about the feature database are provided in Appendix A.1. Notably, all features are stored in a manner consistent with our APK characterization taxonomy. Each directory corresponds to a specific feature source, and the features of a given APK are distributed accordingly — for instance, Hardware Components and Application Components under Manifest; API Calls, Opcodes, and Program Graphs under Dex; Native Functions under Library; and Image Resources under Resource. Such structured storage enables developers to easily query and extract the required features without re-extracting them from the original APKs, greatly accelerating the development of new detectors. In addition, the modular design allows new feature types to be seamlessly integrated into the framework.

The *preprocessor* retrieves and encodes features before they are fed into the ML model. This module can be customized to support different detectors, providing flexibility in feature selection and usage. Leveraging the structured organization of the *feature database*, users can easily select and combine features from multiple sources to construct customized feature vectors. For instance, when designing a new detector, one can quickly retrieve a subset of features at the APK-characterization level using the *preprocessor*. To replicate a detector that relies on API calls and permissions, the *preprocessor* can be configured to extract these features from the Dex and Manifest categories, respectively, and encode them into a unified feature vector.

The *ML model* module integrates a wide range of commonly used machine learning models, such as RF, SVM, KNN, MLP, LSTM, CNN, GNN, and AE. Each model offers user-friendly interfaces for training, testing, and evaluation, and the module supports easy customization and extension — for example, adding new neural network architectures. To develop a new detector, users only need to customize the *preprocessor* to select the required features from the *feature database* and feed them into the chosen model within the *ML model* module. If certain features or models are not yet supported, the framework can be readily extended by adding new feature types to the *feature database* and incorporating additional learning models into the ML module.

FrameDroid provides configurable experimental settings to support different evaluation scenarios. Parameters, such as training sample size, goodware-to-malware ratios, and temporal distribution of samples, are all adjustable. The framework offers fine-grained control over these parameters and supports checkpointing during both training and evaluation. This flexibility facilitates app selection and scenario construction, ensuring comprehensive and accurate evaluations. For example, users can configure the goodware-to-malware ratio to assess model performance under varying malware prevalence. They can also adjust the temporal distribution of apps to simulate malware evolution, which is essential for evaluating model robustness. Furthermore, with flexible

training-sample configuration and organized checkpoint management, FrameDroid enables incremental training, allowing users to incorporate new training samples and continue training from existing models.

FrameDroid also integrates an APK obfuscator [15], which can produce various types of obfuscated samples, *e.g.*, identifier renaming, resource encryption, code modification, and invocation reflection. Obfuscation is applied to APKs before feature extraction to generate their obfuscated variants. The different obfuscation types can be applied as independent operations: one can first apply one type of obfuscation and then apply another to an already obfuscated APK. This capability is essential for evaluating the model's resilience to obfuscated malware. Moreover, this framework incorporates an adversarial sample generation mechanism from AndroidHIV [27], enabling the assessment of model resilience to adversarial attacks. Specifically, adversarial samples are generated by perturbing the feature vectors of original samples, and these perturbed samples are then used to evaluate the robustness of the trained models. The perturbation process ensures that the functionality of the original apps is preserved. Further details can be found in AndroidHIV [27]. Such flexibility and adaptability make FrameDroid a versatile tool for developing and evaluating ML-based Android malware detectors.

FrameDroid Implementation. FrameDroid is developed in 17K lines of Python code. To ensure consistency and mitigate biases from different feature extraction toolchains and learning frameworks, we adopt a standardized setup across all Android malware detectors. For feature extraction, we use Androguard [1] to disassemble APK files and derive features such as permissions, intents, and program graphs. LibRadar [8] helps in identifying third-party libraries within the applications, while Angr [2] is used for analyzing native libraries and capturing essential features like opcodes and API calls. When it comes to machine learning models, the scikit-learn library is our choice for traditional ML algorithms like SVM, KNN, DT, and RF. On the other hand, for deep learning architectures such as CNN, GNN, and AE, we resort to Pytorch [9]. This uniform approach ensures a balanced evaluation, concentrating purely on the uniqueness and performance of each method.

Potential usage and availability of FrameDroid. As discussed above, FrameDroid is designed with a modular and flexible architecture, making it easy to extend and customize to meet diverse research and evaluation requirements. The framework also incorporates a comprehensive app dataset that spans a wide range of Android applications across different time periods. Researchers and practitioners can readily leverage FrameDroid to develop new ML-based Android malware detectors, evaluate their performance under diverse scenarios, and compare them with existing approaches in a fair and consistent manner. For example, when designing a new detector based on API calls and permissions, users can easily retrieve these features from the *feature database* and feed them into their proposed model. Likewise, existing detectors can be readily reproduced by appropriately configuring the *preprocessor* and selecting the corresponding models from the *ML model* module, enabling systematic and reproducible comparisons. It has already been adopted by several academic and industrial organizations across multiple countries, including the United States, Singapore, and China, for the development and evaluation of ML-based Android malware detectors. This level of adoption demonstrates the practicality and effectiveness of FrameDroid in supporting malware analysis research.

4 Representative Approach Analysis

To understand the state of ML-based Android malware detection, a quantitative analysis of current work is indispensable. While an ideal scenario is evaluating as many approaches as possible, conducting an exhaustive examination of each method is impractical due to the vast amount of existing literature. Moreover, it is important to understand that, despite vast publications in this area achieving promising results, many of them share similar techniques (*e.g.,* similar neural networks), and novel solutions are comparatively fewer. Thus, our analysis focuses on methods that represent the broad spectrum and depth of advancements in the field. That is, we aim to select representative approaches and utilize our framework to re-implement and evaluate them. Specifically, in this section, we outline the selected approaches and provide a comparative analysis to highlight their experimental

Table 1. A summary of our selected approaches regarding APK Characterization, Feature Representation, and ML Models. ● indicates that the APK feature is utilized in feature engineering, while ○ is the opposite.

Selected Approach	APK Characterization										Feature Representation	ML Models
	Hardware Component	Application Component	Intent	Permission	API Call	Byte Code	Opcode	Code String	Program Graph	Resource Information		
Drebin [17]	●	●	●	●	●	○	○	●	○	○	Categorical	SVM
MamaDroid [65]	○	○	○	○	●	○	○	○	●	○	Graph-based	RF
Mclaughlin et al. [67]	○	○	○	○	○	○	●	○	○	○	Image-based	CNN
HinDroid [50]	○	○	○	○	●	○	○	○	●	○	Graph-based	SVM
DeepRefiner [102]	●	●	●	●	○	●	○	○	○	●	Text-based	LSTM
Kim et al. [56]	●	●	●	●	●	○	●	●	○	○	Categorical	MLP
MalScan [96]	○	○	○	○	●	○	○	○	●	○	Graph-based	KNN
SDAC [99]	○	○	○	○	●	○	○	○	●	○	Categorical	SVM
HomDroid [97]	○	○	○	○	●	○	○	○	●	○	Categorical	KNN
Xmal [94]	○	○	○	●	●	○	○	○	○	○	Categorical	MLP
RAMDA [61]	○	○	●	●	●	○	○	○	○	○	Categorical	AE
MSDroid [48]	○	○	○	●	●	○	●	●	●	○	Graph-based	GNN

While multiple ML models may be utilized in individual approaches [65, 96, 97], we report the model that yields the best effectiveness (*e.g.*, F1-score).

designs. We then quantitatively evaluate these approaches in Section 5, shedding further light on the current state of ML-based Android malware detection.

4.1 Selection Criteria

Cover various communities. Android malware detection stands as an interdisciplinary domain, drawing contributions from diverse communities, including software engineering, security, and machine learning. However, a notable separation is observed — the solutions in one community often only compare with others from the same community — which hinders potential collaborative advancements. For instance, methods like [50, 105] from one community are often evaluated in isolation, complicating direct performance comparison. To bridge the gap, we incorporate approaches from leading venues across a broad range of communities.

Explore an extensive spectrum of techniques in the detection pipeline. As identified in Sec. 2, a wide range of technique combinations exists across different phases of the detection pipeline. This study explores as many techniques as possible in each phase, including various mixes of APK features, feature representations, and ML models. Such a comprehensive investigation enables us to gain a thorough understanding of the entire workflow of ML-based Android malware detection.

Reflect research progress. In consideration of the evolving research landscape, where new methodologies continually emerge, we place emphasis on approaches that introduce cutting-edge techniques. Particularly, we prioritize methods that either propose novel feature representations or employ innovative learning architectures, or achieve remarkable performance in the field.

Emphasize representative approaches over specific papers. Many approaches share similar techniques, extracting patterns from analogous feature sets (*e.g.,* program graphs) with similar ML models such as different variants of GNNs. Analyzing these methods could lead to redundancy and provide limited insights. Thus, we focus on distinct and representative strategies that offer more significant contributions.

4.2 Selected Approaches

In Section 2, we have presented recent advancements in ML-based Android malware detection by dissecting the detection pipeline into three phases: APK characterization, feature representation, and ML modeling. Adhering

to the selection criteria, we identify 12 representative approaches from hundreds of available solutions to analyze the current state of this field. During the selection process, to foster collaboration among different fields, we prioritize the approaches published in leading venues from three communities, such as ASE, TOSEM from software engineering, NDSS, TIFS from security, and KDD, WWW from machine learning. Specifically, the selected approaches include 3 from software engineering, 7 from security, and 2 from machine learning. A detailed summary about sources and publication years of these approaches is provided in Table 15. We also ensure that the selected approaches cover almost all the APK characterization, feature representation, and ML models discussed in Section 2. As Table 1 shows, they are carefully chosen to represent a broad range of techniques employed in the detection process, including 10 APK features, 4 feature representations, and 8 ML models. Additionally, the selected approaches are distinguished either by their promising performance or by introducing novel techniques. For instance, both MsDroid [48] and EFCG [23] are recent works that introduce GNNs to detect malicious patterns in APKs. We highlight MsDroid as it better represents apps' graph structures via aggregating more channel information and currently stands as the state-of-the-art in Android malware detection using GNNs. Importantly, we also make a concerted effort to ensure that the selected solutions are not variants of existing ones. For example, several methods [34, 50, 105] utilize heterogeneous graphs to model APKs, we select HinDroid [50] because it introduces a novel feature representation that encodes diverse API call relationships while achieving performance comparable to other methods. To better understand the selected representative approaches, we introduce them in the following section.

Drebin. Drebin [17] first collects APK features, such as hardware components, permissions, intents, and API calls, using static analysis. These features are then converted into a binary vector to signify their existence or not. An SVM is then trained to detect Android malware.

MamaDroid. MamaDroid [65] pioneers the use of Markov chains to model app behavior. It constructs a Markov chain over the sequence of abstracted API calls invoked by an app and computes the transition probabilities between these calls as features, which are then used by an RF classifier to detect Android malware.

Mclaughlin et al. This technique [67] presents the leading edge in image-based Android malware detection. By transforming opcode sequences into images via a one-hot encoding, it leverages a CNN model to classify them as benign or malicious samples.

HinDroid. Hou et al. [50] introduce a novel approach by representing API calls as a structured heterogeneous information graph. This approach accounts for the inter-relationships among API calls, such as their presence in the same code block. It then captures apps' semantics with meta-path techniques [86]. A multi-kernel SVM is further applied to recognize malicious patterns.

DeepRefiner. DeepRefiner [102] designs a two-layer malware detection system. Initially, it feeds features like hardware components, permissions, and resources into an MLP to detect most malware. For ambiguous cases, it further interprets APK bytecodes as text sequences and employs an LSTM model to capture the method-level and application-level semantics behind app behaviors.

Kim et al. Kim et al. [56] use multi-modal learning to detect malware, aggregating various features, such as intent and API calls. Distinct MLPs are initially utilized to process individual features independently. Subsequently, a unified MLP integrates the outputs from the preceding models, offering a consolidated decision on malware identification.

MalScan. This study [96] treats program graphs as social networks, where API calls are treated as nodes, and the relationships between them are presented as edges. The system further evaluates the centrality of sensitive API calls in the graph to derive features and then feeds them into a KNN model to detect malware.

SDAC. It attempts to cluster API calls based on their contextual information extracted from API call sequences [99]. These resulting clusters act as features to represent APKs. An SVM model is then used to capture malicious patterns.

Table 2. A comparative study of our selected approaches based on their experimental setup, efficiency evaluation, robustness evaluation, artifact release, and toolchain. — denotes the absence of the statistics in the literature. ● indicates that both feature encoding and ML modeling were evaluated for efficiency, ◐ indicates that only ML modeling was evaluated, and ○ indicates that the efficiency was not evaluated. Malware Ratio refers to the proportion of malware samples in a testing set.

Selected Approach	Experimental Setup				Efficiency Evaluation	Robustness Evaluation			Artifact Release	Tool Chain
	Dataset Size	Time Span	Train: Val:Test	Malware Ratio		Evolution	Obfuscation	Adversarial Sample		
Drebin [17]	129,013	2010-2012	2:0:1	4%	●	✘	✘	✘	✔	Androguard
MamaDroid [65]	43,940	2010-2016	9:0:1	50%	●	✔	✘	✘	✔	Soot
Mclaughlin et al. [67]	27,395	—	—	50%	◐	✘	✘	✘	✔	BackSmali
HinDroid [50]	2,334	2017-2017	4:0:1	60%	◐	✘	✘	✘	✘	APKTool
DeepRefiner [102]	110,440	—	**8:1:1**	57%	●	✘	✔	✔	✘	APKTool
Kim et al. [56]	41,260	—	**3:1:1**	50%	◐	✘	✔	✘	✘	APKTool
MalScan [96]	30,715	2011-2018	9:0:1	50%	●	✔	✘	✔	✔	Androguard
SDAC [99]	70,142	2011-2016	8:0:2	50%	●	✔	✔	✘	✘	FlowDroid
HomDroid [97]	8,198	—	9:0:1	40%	●	✘	✘	✘	✘	Androguard
Xmal [94]	35,690	—	7:0:3	43%	○	✘	✘	✘	✔	Androguard
RAMDA [61]	58,483	—	19:0:1	50%	○	✘	✘	✔	✔	APKTool
MSDroid [48]	81,790	2010-2015	4:0:1	37%	○	✔	✔	✘	✔	Androguard

HomDroid. The method [97] focuses on suspicious parts of malware by calculating the homophily in its program graph. From the malicious subgraphs, it derives two key features: (1) the presence of sensitive API calls, and (2) the number of sensitive triads. These features are then fed into a KNN model to detect malware.

Xmal. Xmal [94] utilizes MLPs to distill information from extracted API calls and permissions for Android malware detection. It further integrates an attention mechanism to highlight the most informative features. This attention-based MLP not only achieves promising results but also offers an interpretation of the model.

RAMDA. This detector [61] is the SOTA approach that employs Autoencoder to derive a resilient representation of APKs with features such as API calls and intents. The representation is fed into an MLP to detect malware.

MSDroid. MSDroid [48] is the cutting-edge in utilizing GNN to detect malware. Initially, it breaks down the program graph into subgraphs rooted at sensitive API calls. Then, it leverages a GNN to capture essential semantics from the graph representations for malware detection.

4.3 Comparative Study

Analyzing these approaches from various angles provides insightful lens for understanding the current advancements and challenges in ML-based Android malware detection. In this section, we conduct a comparative review of the selected approaches, focusing on the following three critical dimensions. (a) *Effectiveness* refers to the ability of an approach to accurately identify malware under various circumstances, such as different dataset sizes and goodware-to-malware ratios; (b) *Robustness* assesses the methods' resilience, especially in response to challenges like malware evolution; (c) *Efficiency* reflects the computational overhead incurred during APK processing and ML modeling. Building on this, we further emphasize the importance of employing a unified framework to evaluate the effectiveness, robustness, and efficiency of ML-based Android malware detection techniques.

Effectiveness. Effectiveness is the most important criterion for any detection technique, and all the selected approaches evaluate this aspect. However, the experimental setup varies dramatically across these approaches, making direct comparisons challenging. As indicated in Table 2, the experimental settings — dataset size, time

span, dataset partition, and malware ratio — demonstrate large variations across studies. It is well-established that a positive correlation exists between the training data size and ML model performance. The training data size in these approaches ranges from 2,334 [50] to 129,013 [17], making it difficult to compare their effectiveness. The datasets' temporal span further complicates the evaluation, as Android malware evolves over time and the features extracted from APKs change accordingly. Another point is the absence of a validation set [48, 61, 94]. This oversight is alarming, raising concerns about potential over-fitting and over-optimistic performance indicators. In the wild, the ratio of malware to benign apps is notably imbalanced, with malware accounting for around 10% of cases [77]. However, this ratio in testing datasets significantly varies across different studies (*e.g.,* from 4% [17] to 60% [50]), which makes it difficult to reveal the true effectiveness of these approaches.

Robustness. Robustness stands as a pivotal criterion for any methodology. Within Android malware detection, this robustness typically encompasses an approach's capacity to counteract malware evolution, obfuscation strategies, and adversarial attacks [77]. Specifically, detectors routinely operate in hostile and dynamic contexts [21], where malware constantly evolves to evade detection. Also, obfuscation techniques have been widely adopted by attackers to conceal their malicious operations [15]. Additionally, the inherent susceptibility of ML models to adversarial attacks [18, 59] complicates the detection process. We observe that these selected approaches miss one or more of the challenges, as shown in Table 2. This omission complicates the assessment of their true effectiveness in real-world deployments.

Efficiency. To measure a new malware detector, the importance of efficiency stands parallel to effectiveness. As Android apps grow in size and complexity, the time and computational resources required for APK processing and ML modeling could substantially rise. However, as Table 2 shows, not every approach evaluates the efficiency of these two parts. Additionally, understanding how efficiency shifts when dealing with APKs at various times is vital to ensure detectors' sustainability and long-term utility; unfortunately, this is not considered by any of the selected approaches.

Artifact Release and Toolchain. Table 2 also reports whether the selected approaches make their artifacts publicly available and details the specific toolchains they employ. We observe that nearly half of these approaches do not release their artifacts, which poses significant challenges for reproducibility and subsequent research. Furthermore, the toolchains they use are diverse, including Androguard [1], APKTool [3], and BackSmali [4]. We know that different toolchains may yield varying analysis results for the same APK due to differences in their static analysis capabilities [17, 65]. As a result, it is difficult to fairly compare the effectiveness and efficiency of these approaches when they rely on heterogeneous toolchains. To further investigate this issue, we conduct an experimental analysis in Section **??**.

Combining the aforementioned analyses, there is a pressing need for a general-purpose framework that standardizes the entire development pipeline, from feature extraction and model construction to training and deployment. Such a framework should also support diverse and configurable evaluation scenarios for ML-based Android malware detectors, enabling fair comparison across approaches, improving reproducibility, and facilitating large-scale empirical studies.

5 Quantitative Analysis

One of our key contributions is a comprehensive quantitative analysis of the selected approaches. This analysis aims to assess and disentangle the effects of various experimental settings — such as data size, goodware-to-malware ratios, and the presence of adversarial attacks — on the performance of these representative detectors. By systematically varying these conditions, we can observe how current ML-based Android malware detection methods behave under different scenarios and identify key factors that contribute to an effective detector. Specifically, we re-implement the 12 representative approaches outlined in Section 4 within our general-purpose

Table 3. Evaluation Dataset Statistics of the Primary Dataset. The unit used for measuring APK size is megabytes (MB).

Year	Malicious	Benign	M+B	M/(M+B)	Avg.Size
2011	2,085	17,878	19,963	10.4%	2.26
2012	2,137	18,687	20,824	10.3%	3.58
2013	2,182	18,631	20,813	10.5%	5.21
2014	2,346	20,142	22,488	10.4%	6.91
2015	2,369	20,643	23,012	10.3%	9.52
2016	2,390	21,292	23,682	10.1%	12.26
2017	2,389	21,006	23,395	10.2%	16.24
2018	2,326	20,099	22,425	10.4%	16.62
2019	2,345	20,260	22,605	10.4%	17.27
2020	2,301	19,802	22,103	10.4%	16.65
Total	22,870	198,440	221,310	10.3%	10.86

framework, FrameDroid (implementation details are provided in Appendix A.2). We then evaluate their effectiveness (Section. 5.2), robustness (Section. 5.3), and efficiency (Section. 5.4) on a large-scale, long-duration dataset (Section. 5.1) tailored for this study. Beyond these experiments, we also investigate additional aspects, such as the influence of reverse-engineering toolchains and model selection given specific feature sets, to derive deeper insights into the current state of Android malware detection.

5.1 Dataset

Dataset Construction Principles. To conduct an evaluation that is representative of real-world scenarios and enables a comprehensive, multi-dimensional assessment, it is essential that the dataset satisfies the following criteria [77].

- *Market Diversity*: Android apps should be collected from multiple app stores to ensure comprehensive representation of the ecosystem. We consider various sources, including Google Play, VirusShare, and malware repositories.
- *Exclusion of Grayware*: Grayware should be excluded to prevent skewing the results, as its uncertain nature can complicate the classification of apps. To filter out grayware, we use the number of positive antivirus alerts (denoted as p) derived from VirusTotal [11] as a criterion. Following previous studies [70, 77], apps with $p \geq 4$ are categorized as malicious, while those with $p = 0$ are considered benign.
- *Temporal Distribution*: Malware evolution is a critical factor affecting detection performance; thus, the dataset should span a wide range of years to capture this dynamic.
- *Realistic Malware Ratio*: The dataset should reflect the actual goodware-to-malware ratio observed in the wild to ensure that the evaluation is grounded in real-world conditions. Specifically, we follow previous studies [28, 77] and set the malware ratio to 10%, to establish a realistic evaluation environment.
- *Deduplication*: To avoid bias in the evaluation, duplicate apps should be removed from the dataset. We first perform hash-based deduplication to eliminate exact copies of the same APK. We then filter out apps that share the same package name and version code, as they are likely to be identical or highly similar. Finally, we inspect the extracted features, such as the number of permissions and API calls, to identify potential duplicates that may have been repackaged or slightly modified.

Following these principles, we construct two datasets: a primary dataset (Section 5.1.1) used for the main evaluation, and an additional dataset (Section 5.1.2) employed for further validation and complementary experiments.

Table 4. Data distribution of four sub-datasets derived from primary dataset. M/N indicates M benign and N malicious apps.

Training	Validation	Testing	Alias
14,400/1,600	1,800/200	1,800/200	❶
	1,000/1,000	1,000/1,000	❷
8,000/8,000	1,800/200	1,800/200	❸
	1,000/1,000	1,000/1,000	❹

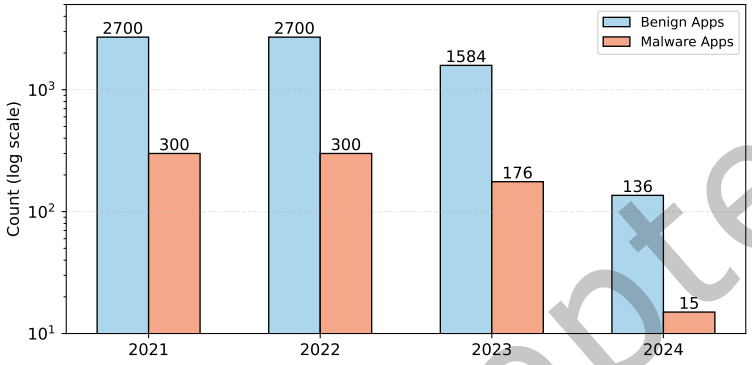


Fig. 4. Data statistics of apps collected from 2021 to 2024. The y-axis uses a logarithmic scale.

5.1.1 ***Primary Dataset.*** To build the main evaluation dataset, we take AndroZoo [13] as our app source, which is a growing collection of apps from different platforms, like Google Play, PlayDrone, and AppChina. As for the temporal distribution, we set the time span as ten years (*i.e.,* from 2011 to 2020) for a comprehensive evaluation, following previous studies [28, 38]. To mimic actual conditions, we select apps based on a monthly time window, ensuring that around 2,000 apps are collected each month, with a goodware-to-malware ratio aligning with the estimated 9 : 1 in the wild [77]. Additionally, to ensure the quality of the dataset, we exclude apps that cannot be parsed by AndroGuard [1] because we cannot extract the required features for malware detection. Finally, we obtain a dataset of 221,310 Android apps, i.e., 22,870 malicious and 198,440 benign apps, as summarized in Table 3. We employ this primary dataset for the main evaluation of the selected approaches in effectiveness, robustness, and efficiency.

Settings. To further support specific evaluation scenarios, we construct four sub-datasets with different configurations derived from the primary dataset. For each sub-dataset, apps are randomly sampled to form training, validation, and testing sets, and their distributions are summarized in Table 4. Importantly, the apps in these sub-datasets are sampled from different years to ensure comprehensive temporal representation. For instance, for ❶, each year contributes 1,440 benign and 160 malicious apps for training, 180 benign and 20 malicious apps for validation, and the same number for testing. Duplicate apps are also avoided across these sets. We use these sub-datasets to investigate the effectiveness of the selected representative approaches under different settings, such as goodware-to-malware ratio and APK features.

5.1.2 ***Additional Dataset.*** As a complementary dataset, we collect Android apps from different sources, *i.e.,* AndroZoo [13] and VirusTotal [11], spanning the years 2021 to 2024, following the same principles outlined above. During this process, we observe that the number of malware samples in these years is relatively small, making it challenging to support a thorough evaluation [16]. To enrich the dataset, we therefore additionally gather malware samples from VirusShare [10], a well-known repository that aggregates a large collection of

Table 5. The effectiveness (F1 and Accuracy) of the approaches across varied goodware-to-malware ratios in training and testing sets.

Selected Approach	B:M=9:1 in Tr (❶ - ❷)				B:M=1:1 in Tr (❸ - ❹)			
	B:M=9:1 in Ts		B:M=1:1 in Ts		B:M=9:1 in Ts		B:M=1:1 in Ts	
	F1	Acc.	F1	Acc.	F1	Acc.	F1	Acc.
Drebin	0.722	**0.948**	0.791	0.823	0.650	0.901	0.918	0.917
MamaDroid	0.661	0.945	0.693	0.763	0.650	0.902	0.895	0.897
Mclaughlin et al.	0.714	0.946	0.799	0.828	0.682	**0.924**	0.916	0.916
HinDroid	**0.731**	0.943	0.819	0.842	**0.701**	0.924	**0.925**	**0.925**
DeepRefiner	0.657	0.932	0.776	0.809	0.667	0.918	0.881	0.881
Kim et al.	**0.782**	**0.952**	**0.907**	**0.912**	**0.753**	**0.941**	**0.938**	**0.937**
MalScan	0.684	0.939	0.793	0.823	0.587	0.877	0.880	0.880
SDAC	0.522	0.916	0.627	0.720	0.524	0.850	0.845	0.844
HomDroid	**0.734**	**0.949**	0.816	0.841	**0.701**	**0.925**	0.912	0.914
Xmal	0.698	0.942	0.826	**0.847**	0.674	0.916	**0.923**	**0.924**
RAMDA	0.636	0.905	**0.841**	**0.852**	0.510	0.829	0.871	0.865
MSDroid	0.648	0.919	**0.834**	0.828	0.522	0.853	0.867	0.858

Tr: training set, *Ts:* testing set. ❶ - ❹ correspond to the four scenarios in Table 4. The top 3 results for each scenario are highlighted in bold.

malware from various sources. We maintain a realistic goodware-to-malware ratio of 9:1 in this additional dataset. Figure 4 summarizes its statistics, which in total contains 7,911 apps, including 791 malicious and 7,120 benign samples. We note that the number of malicious samples in this additional dataset is still relatively small compared to the primary dataset. Nonetheless, it is sufficient to support our complementary experiments, which primarily aim to validate the findings from the main evaluation, particularly regarding the effectiveness (*e.g.,* whether detection performance changes significantly) and efficiency (*e.g.,* memory usage) of the selected approaches.

5.2 Effectiveness

In this section, we examine how the detection effectiveness of the selected approaches varies under different experimental settings, with the goal of identifying the key factors that contribute to an effective malware detector. Specifically, we focus on the following aspects: goodware-to-malware ratio, training set size, APK feature choices, and model selection. These factors are pivotal because they influence the quality of the training data and the model's ability to extract relevant information, both of which are critical to the performance of ML-based detectors. All experiments in this section are conducted on the primary dataset (Section 5.1).

Goodware-to-malware ratio. The effectiveness of ML-based Android malware detectors is heavily influenced by malware distributions in the training and testing sets. Here, we consider two ratios, *i.e.*, 10%, and 50%, in both training and testing sets. These choices are inspired by the estimated 10% in the wild [28, 77] and the 50% widely used in previous studies [61, 65, 96]. The ❶ - ❹ in Table 4 present these four scenarios, where ❶ and ❷ have a 10% malware ratio in training sets, while ❸ and ❹ have a 50% malware ratio.

Table 5 summarizes the results in terms of F1-score and Accuracy. The results are arranged from left to right, corresponding to scenarios ❶ through ❹. We observe that all approaches achieve promising performance in all metrics under scenario ❹, which is characterized by a 1 : 1 goodware-to-malware ratio in both the training and testing sets. These findings are consistent with the results reported in the original papers, confirming the validity of our re-implementations. This demonstrates that these approaches can effectively detect Android malware under ideal conditions, where goodware and malware are balanced. However, when the malware proportion is reduced to 10% (❶), there is a pronounced decline in F1-score across all methods. It is evident that many existing approaches experience performance degradation when evaluated under realistic malware ratios. This is primarily attributed to the scarcity of malware samples in the training data, which constrains the model's ability to learn discriminative malware semantics. Moreover, when the malware ratio in the test set is increased from 10% to 50% (❷ and ❹), we observe a consistent improvement in F1-scores across all methods. This trend is expected, as a higher prevalence of malware samples in the test set increases the likelihood that models will encounter patterns similar to those seen during training, enhancing their predictive performance. To more clearly reveal the performance differences between deep learning (DL)-based and traditional machine learning (TML)-based approaches, we conduct a coarse-grained, averaged analysis of the results, as averaging can partially mitigate the influence of the differing feature combinations employed by both types of approaches. We observe an interesting phenomenon: DL-based approaches appear to outperform TML-based methods when the testing malware ratio is 50% (❷ and ❹) — almost all the top three results are achieved by DL-based methods. While in more realistic scenarios (❶ and ❸), DL-based approaches do not exhibit a pronounced advantage over TML-based methods. One possible explanation is that DL-based methods are better equipped to capture the complex semantics underlying app behaviors, but require more malware samples to effectively distill such semantics, which tend to be more prominent in datasets with a higher malware ratio.

Table 6 provides Precision and Recall values for these scenarios. When the training set ratio is fixed, we observe that the malware detection performance (in terms of Recall) does not change significantly as the testing set ratio varies, as shown by pairs ❶-❷ and ❸-❹ in Table 6. In contrast, when the testing set ratio is fixed, we find that the Recall improves as the training set ratio increases from 10% to 50%, as illustrated by pairs ❶-❸ and ❷-❹ in Table 6. This indicates that having a higher proportion of malware samples in the training set enhances the model's ability to learn effective malicious patterns, thereby improving detection accuracy.

Training set size. The size of the training set is a critical determinant of the effectiveness of ML classifiers [83], as larger, high-quality datasets typically provide richer semantic information about malware, which is essential for learning effective classifiers. However, obtaining such large, high-quality training sets is often infeasible due to the substantial costs associated with data collection and labeling. In this experiment, we experiment with different training set sizes to simulate varying levels of semantic richness (*i.e.,* larger datasets are assumed to encompass more diverse malware semantics) and analyze their impact on classifier performance. Specifically, we consider two training set sizes, 50% and 10% of the original training set size, while maintaining the malware ratio at 10%.

Figure 5 shows how these detectors' performance changes when the training set size or semantic richness is reduced. We normalize the F1-scores of these methods based on their performance obtained with the entire original dataset. The detailed results are provided in the Appendix A.4. The figure clearly shows that reducing the training set size leads to performance degradation across all approaches, underscoring more data enhances capturing malicious patterns. Interestingly, Mclaughlin et al. and DeepRefiner display a greater sensitivity to training set size compared to others. One explanation is that, unlike other methods that heuristically select features from apps, these two approaches take the original apps' bytecode as input and automatically extract semantics, requiring more data to learn patterns from raw data as opposed to hand-crafted features. This finding highlights that the effectiveness of malware detectors is closely tied to the quality of the training data and the

Table 6. The effectiveness (precision and recall) of the approaches across varied goodware-to-malware ratios.

Selected Approach	B:M=9:1 in Tr (❶ - ❷)				B:M=1:1 in Tr (❸ - ❹)			
	B:M=9:1 in Ts		B:M=1:1 in Ts		B:M=9:1 in Ts		B:M=1:1 in Ts	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
Drebin	0.776	0.675	0.964	0.671	0.501	0.925	0.906	0.930
MamaDroid	0.863	0.535	0.984	0.535	0.506	0.910	0.905	0.886
Mclaughlin et al.	0.758	0.675	0.958	0.685	0.584	0.820	0.908	0.925
HinDroid	0.722	0.740	0.956	0.717	0.578	0.890	0.914	0.937
DeepRefiner	0.663	0.650	0.944	0.659	0.562	0.820	0.883	0.878
Kim et al.	0.717	0.860	0.960	0.859	0.644	0.905	0.922	0.954
MalScan	0.704	0.665	0.954	0.678	0.442	0.875	0.878	0.881
SDAC	0.632	0.480	0.908	0.511	0.379	0.850	0.835	0.838
HomDroid	0.755	0.714	0.963	0.708	0.583	0.879	0.926	0.898
Xmal	0.728	0.670	0.953	0.729	0.549	0.875	0.927	0.920
RAMDA	0.516	0.830	0.910	0.781	0.357	0.890	0.829	0.918
MSDroid	0.563	0.760	0.805	0.866	0.387	0.800	0.820	0.919

Tr: training set, *Ts:* testing set. ❶ - ❹ correspond to the four scenarios in Table 4. The top 3 results for each scenario are highlighted in bold.

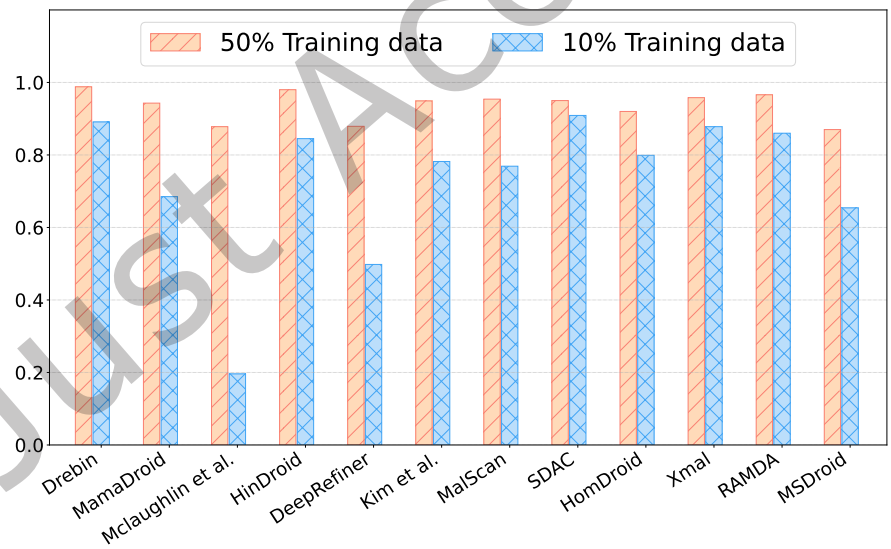


Fig. 5. Effectiveness of the selected approaches using different sizes of training data.

semantic richness it provides. Furthermore, these findings suggest that carefully selected features that better capture malware semantics are crucial for effective detection, especially when training data is limited. In contrast, models that learn directly from raw data often struggle to perform well under such data-scarce conditions.

Table 7. The impact of APK features on the effectiveness of Drebin, Kim et al., Xmal, and RAMDA.

Feature Combination	Drebin		Kim et al.		Xmal		RAMDA	
	F1	Acc.	F1	Acc.	F1	Acc.	F1	Acc.
Original	**0.722**	**0.948**	0.726	0.944	**0.698**	**0.942**	**0.636**	**0.905**
w/o hardware	0.717	0.947	0.728	0.945	N/A	N/A	N/A	N/A
w/o app-intent	0.622	0.936	**0.776**	**0.952**	N/A	N/A	0.428	0.845
w/o permission	0.698	0.945	0.692	0.939	0.566	0.926	0.526	0.882
w/o api call	0.691	0.944	0.699	0.942	0.639	0.930	0.482	0.880
w/o opcode	N/A	N/A	0.724	0.942	N/A	N/A	N/A	N/A
w/o code string	0.702	0.944	0.725	0.945	N/A	N/A	N/A	N/A

w/o means without. N/A denotes features that are not used in the original work.

APK feature. Android malware detectors often leverage diverse features to enhance detection performance [17, 56, 61]. The reason is that each feature contributes to characterizing APKs, encoding their unique semantics. One question arises: does the incorporation of more features necessarily enhance a detector's performance? To investigate this, we remove individual features from the original feature set to assess the resulting performance. For this experiment, it is essential that the required features of the chosen approaches are decomposable, allowing the sequential removal of individual features. Accordingly, we spotlight Drebin, Kim et al., Xmal, and RAMDA, owing to their decomposable feature sets. We exclude methods whose features are highly intertwined. For instance, MamaDroid and MalScan rely on specific API calls to extract features from program graphs, making it challenging to remove individual features — if API calls are removed, the graph features will also be removed.

Table 7 presents the outcomes of this experiment. From the table, we observe that Drebin, Xmal, and RAMDA have performance degradation when one feature is removed. This is intuitive, that each feature contains unique semantics describing the APKs and contributes to the overall effectiveness of a malware detector. Interestingly, Kim et al. deviate from this trend. In fact, even after omitting several features, its performance exceeds the original results. For example, removing the app-intent features, the F1-score improves from 0.726 to 0.776, and Accuracy increases from 0.944 to 0.952. These two observations underscore: (i) each feature contains unique semantics describing the APKs and contributes to the overall effectiveness of a malware detector, and (ii) in some cases (*e.g.,* specifc models or feature combination style), simply expanding the feature set does not guarantee enhanced performance. This phenomenon underscores the importance of evaluating and justifying the inclusion of each feature in a malware detector based on the semantics.

We further examine the impact of feature combinations on the approach proposed by Kim et al. by varying the number of training epochs, in order to verify whether the observed phenomenon persists across different model configurations. Specifically, we set the training epochs to 50, 100, 150, and 200, and repeat the app-intent removal experiment under each setting to assess result consistency. As shown in Table 8, removing the app-intent feature consistently improves performance across all training epochs, in terms of both F1-score and accuracy. This demonstrates that the observed behavior is robust and not an artifact of a particular training configuration or convergence state. These findings highlight an important insight: under certain conditions—such as specific classifiers or feature-combination strategies—the inclusion of additional features does not necessarily enhance detection performance and may even be detrimental. Consequently, researchers and practitioners should critically

Table 8. Ablation study on the impact of app-intent information across different training epochs.

Method	Metric	50	100	150	200
w/o app-intent	F-Score	0.740	0.776	0.753	0.753
	Accuracy	0.947	0.952	0.950	0.950
original	F-Score	0.725	0.725	0.740	0.738
	Accuracy	0.944	0.944	0.952	0.942

assess the contribution of individual features to overall performance, rather than assuming that incorporating more features will invariably yield better results.

Model type. Given specific features that describe app behaviors, different models can be used to extract semantics and detect malware. The features reflect the richness of the available semantics, while the models determine how effectively these semantics are leveraged. In this section, we select MamaDroid, HinDroid, and MalScan, which share the same feature set and feature representation, as shown in Table 1, to evaluate the impact of different models on the effectiveness of malware detectors. From the results in Table 5 under scenario ❶, we observe that HinDroid outperforms MamaDroid and MalScan. This suggests that given a fixed feature set, different models demonstrate varying capabilities in extracting semantics and detecting Android malware.

Table 9. Performance of common ML models across API and permission feature subsets.

Model	API+Permission		API		Permission	
	Accuracy	F1-score	Accuracy	F1-score	Accuracy	F1-score
RandomForest	0.984	0.929	0.975	0.897	0.973	0.868
SVM	0.978	0.897	0.975	0.897	0.962	0.794
DecisionTree	0.984	0.927	0.965	0.857	0.959	0.819
KNN	0.973	0.865	0.975	0.897	0.954	0.761
MLP	0.981	0.916	0.973	0.886	0.973	0.872

Beyond analyzing existing approaches, we conduct a small-scale empirical study to examine how different learning models perform given a fixed feature set. Specifically, we select two widely used feature types—API calls and permissions—to represent app behavior, and evaluate five representative models commonly adopted in Android malware detection: Random Forest, SVM, Decision Tree, KNN, and MLP. For this experiment, we use the dataset shown in Figure 4 and randomly sample 4,000 apps (3,600 benign and 400 malicious). The dataset is split into training, validation, and testing sets using an $8:1:1$ ratio.

Intuitively, there is no universally optimal model for Android malware detection, as model performance is closely tied to the choice of feature representation. As shown in Table 9, when API calls are used as features, all evaluated classifiers achieve comparable performance. In contrast, when using permissions alone, MLP significantly outperforms the other models. When combining API calls and permission features, Random Forest achieves the best overall performance, with an accuracy of 98.4% and an F1-score of 92.9%. Overall, our results suggest that ensemble-based models (*e.g.*, Random Forest) and deep learning models (*e.g.*, MLP) generally outperform other traditional classifiers (*e.g.*, SVM, Decision Tree, and KNN) in Android malware detection tasks. Consequently, when designing new detectors, these models represent strong and practical baseline choices.

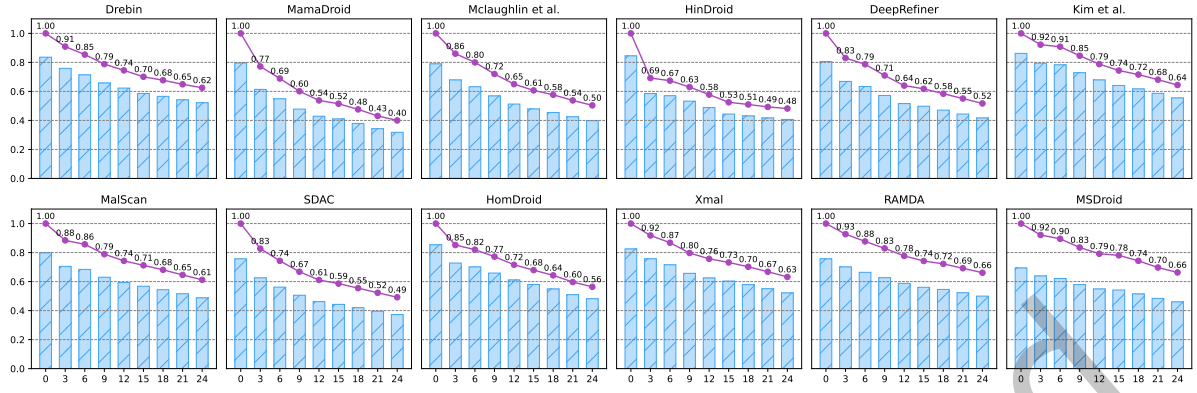


Fig. 6. The performance of the selected techniques against diverse malware evolution periods. Columns display the absolute values of $AUT(F1, N)$. The line charts depict the relative percentage of $AUT(F1, N)$ against $AUT(F1, 0)$.

Summary: The effectiveness of ML-based malware detectors is shaped by various factors, such as APK features and model types. At the core of these influences is the richness of semantics embedded in the features and the models' capability to extract and utilize these semantics.

5.3 Robustness against real-world scenarios

Previous works [38, 59, 77] have started investigating the resilience of detectors when faced with real-world challenges such as malware evolution, obfuscation, and adversarial attacks. Nonetheless, they often focus on a subset of these challenges or employ unrealistic experimental settings, potentially skewing their findings. For instance, TESSERACT [77] explores the impact of malware evolution, while Gao [38] assesses the robustness of detectors using an unrealistic balanced dataset. This section aims to fill these gaps by thoroughly re-evaluating the robustness of the selected approaches in real-word settings, providing a clearer picture of where ML-based malware detection stands today.

Malware evolution. To quantify the impact of evolution on malware detectors, we utilize the $AUT(f, N)$ [77], where f denotes the F1-score of a given approach, and N represents the evolution period. The definition of $AUT(f, N)$ can be found in the Appendix A.3. We set N to 3, 6, 9, 12, 15, 18, 21, and 24 months in this experiment. This metric ranges in (0, 1), where higher values indicate greater resilience to malware evolution. In the study, we adopt a rolling algorithm over the data from 2011 to 2020 to calculate the $AUT(f, N)$. Specifically, for each year, from 2011 to 2020, we first partition the data into training, validation, and testing sets with 8 : 1 : 1. Next, we train models with the training data, validate to get the best model, and evaluate it on the test set to get the F1-score as $AUT(F1, 0)$. Then, the model is applied to test data in the next N months, yielding N F1-scores. It is important to note that the N months of test data may span across different years. For example, if we use data from 2011 as the training set and set N to 15 months, the corresponding test data will include apps collected from January 2012 to March 2013. These scores are further used to calculate the $AUT(F1, N)$. We repeat this process for each year from 2011 to 2020, resulting in ten distinct $AUT(F1, N)$ values for each N, if available. For instance, when N is 3 months, we can retrieve $AUT(F1, 3)$ values for each year from 2011 to 2019, since the test data for 3 months after training in 2020 is not available. By averaging the $AUT(F1, N)$ values sourced from distinct yearly datasets, we chart the outcomes in Figure 6.

It is evident that malware evolution affects the effectiveness of the selected malware detectors. This is expected: as malware evolves over time, its underlying semantics may change, making it more difficult for detectors to capture these behaviors. Overall, most DL-based approaches exhibit stronger resilience to malware evolution than

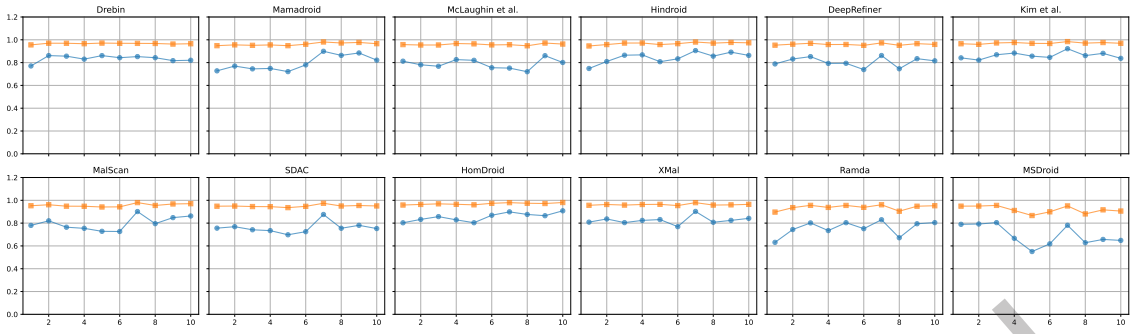


Fig. 7. Performance of the selected detection approaches for each year from 2011 to 2020. Orange markers indicate Accuracy, and blue markers indicate F1-score.

their TML-based counterparts. Specifically, the majority of DL techniques retain around 60% effectiveness even after two years, whereas many TML methods see their F1-scores drop to about 50% over the same period. This disparity can be attributed to the inherent capability of DL models to capture more intricate patterns that TML methods may miss. Interestingly, McLaughlin et al. and DeepRefiner do not perform as well over time. A plausible reason is their reliance on raw bytecode as input, which may contain substantial noise and complicate semantic extraction. To enhance the robustness of ML-based Android malware detectors against malware evolution, two aspects are particularly important: (i) extracting robust features that are less susceptible to temporal changes—for example, Xmal and RAMDA use carefully selected features (such as stable API calls and permissions) and tend to be more resilient to evolution; and (ii) designing models that can effectively capture evolving malware semantics, such as persistent malicious behaviors—for instance, MsDroid employs a graph neural network to model structural relationships among code components, enabling it to better capture these evolving semantics.

Detector stability on different time period datasets. Having examined the impact of malware evolution on detection effectiveness, we now turn to investigating whether the selected approaches can be applied to datasets collected in different time periods. This experiment aims to assess the sensitivity of these approaches to temporal variations in the data. Such an analysis helps determine whether the approaches can be used reliably in practice and whether our findings are generalizable. To cunduct this experiment, we partition the primary dataset into ten subsets based on the year of app collection, spanning from 2011 to 2020. For each year-specific subset, we split the data into training (80%), validation (10%), and testing (10%) sets. Each approach is trained on the training set, with hyperparameters tuned using the validation set, and finally evaluated on the testing set.

Figure 7 illustrates the performance of the selected approaches across different years. We observe that most approaches maintain relatively stable performance over time, indicating that they can be effectively applied to datasets collected at different periods. This stability suggests that the approaches are generally applicable rather than tailored to specific datasets, which in turn supports the claim that our findings are likely to generalize across different time periods.

Obfuscation. Obfuscation techniques often alter an app's code. It can also affect the effectiveness of malware detectors [48]. This part explores the influences of popular obfuscation techniques, *i.e.,* renaming identifiers, encrypting resources, modifying code, and invoking reflection [15], on the selected approaches separately. We apply the obfuscation strategies discussed earlier to the testing set in Table 4(❶). Only the apps that can be successfully obfuscated by all the strategies are included in the obfuscated testing set, which contains 1,303 apps. The effectiveness of the selected approaches, previously trained on the original training set, is then evaluated on this obfuscated testing set.

Table 10 summarizes the results. Overall, most methods exhibit a decrease in performance when subjected to obfuscation. This is expected, as obfuscation can hide malicious semantics, making them more difficult for

Table 10. The F-score of the selected approaches under different obfuscation strategies [15].

Approach \ Obfuscation	Without Obfus.	Rename Identifier	Encrypt Resource	Modify Code	Reflect Invocation
Drebin	0.732	0.702	0.701	0.732	0.732
MamaDroid	0.653	0.274	0.449	0.150	0.461
Mclaughlin et al.	0.750	0.699	0.722	0.175	0.727
HinDroid	0.750	0.750	0.741	0.750	0.735
DeepRefiner	0.692	0.618	0.658	0.297	0.612
Kim et al.	0.795	0.795	0.611	0.795	0.798
MalScan	0.675	0.675	0.675	0.681	0.687
SDAC	0.563	0.538	0.552	0.495	0.552
HomDroid	0.729	0.751	0.706	0.728	0.739
Xmal	0.727	0.727	0.727	0.727	0.727
RAMDA	0.635	0.635	0.635	0.635	0.635
MSDroid	0.672	0.675	0.610	0.356	0.494

detectors to capture. Specifically, the impact of obfuscation on detectors significantly depends on how the detectors utilize APK features. For instance, Xmal and RAMDA show greater robustness against obfuscation. A closer examination reveals that both approaches use API calls as features and rely on manually selected API calls as anchors, checking whether these APIs appear in the code. We further observe that most of these anchor APIs are system APIs, which are less likely to be affected by the obfuscation techniques employed. In other words, the obfuscation does not substantially modify the features used by these detectors. In contrast, approaches such as MamaDroid and MsDroid, which rely heavily on code structure, tend to be more susceptible to code modification. This suggests that the impact of obfuscation on detectors is closely related to the extent to which obfuscation techniques alter the semantics of the features they use. Moreover, comparing MamaDroid and MsDroid—both structure-based—indicates that robustness to obfuscation is also influenced by the model's capability to extract semantics. To better defend against obfuscation, it is therefore crucial to (i) select features that are less likely to be altered by obfuscation techniques and (ii) design models that can effectively extract the underlying semantics from these features.

Adversarial attack. We now utilize the dataset from Table 4(❶) to explore the impact of adversarial attacks on the performance of these selected approaches. It is worth noting that we exclude approaches Mclaughlin et al. and DeepRefiner from this experiment. This is because they truncate features at a certain size, making them easily bypassable. Attackers can embed malicious code in the ignored part to evade detection.

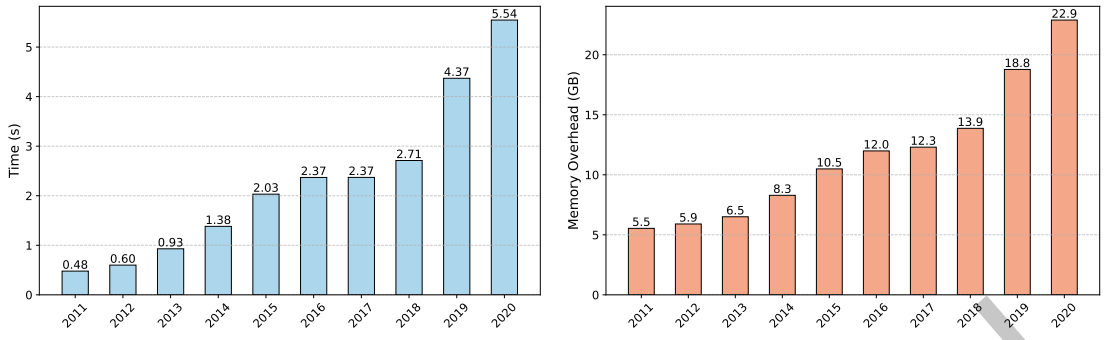
To investigate the impact of adversarial attacks, we employ two primary strategies: Jacobian Saliency Map Attack (JSMA) and Randomized Input (RI). These two techniques are chosen for their simplicity and effectiveness, compromising most ML-based malware detectors, as shown in Table 11. This suggests that more advanced attack strategies could pose even greater threats to these detectors [47, 110]. For JSMA, we first train a substitute model on the training set, and then apply JSMA to generate adversarial samples. Specifically, we utilize a Multi-Layer Perceptron (MLP) as the substitute model for conventional ML-based classifiers (e.g., SVM, KNN, and Random Forest), as MLP can effectively approximate the decision boundaries learned by these classifiers while remaining

Table 11. The robustness of our selected approaches on various adversarial attacks.

Selected Approach	ASR		APR	Selected Approach	ASR		APR
	JSMA	RI			JSMA	RI	
Drebin	1.000	0.237	0.001	SDAC	1.000	0.146	0.001
MamaDroid	0.972	0.187	0.021	HomDroid	0.979	0.303	0.324
HinDroid	1.000	0.068	0.004	Xmal	1.000	0.142	0.013
Kim et al.	1.000	0.000	0.004	RAMDA	0.931	0.300	0.023
MalScan	1.000	0.788	0.001	MSDroid	1.000	0.022	0.013

fully differentiable for gradient-based attacks such as JSMA. For GNN-based detectors (e.g., MsDroid), we adopt a Graph Convolutional Network (GCN) as the substitute model, as GCN captures both node features and edge structures through message passing, and provides a simple yet representative architecture for modeling graph-based malware detectors. During the crafting of adversarial samples, we calculate the Jacobian matrix to identify the most influential features and iteratively modify them until the samples are misclassified by the target model or a maximum number of modifications is reached. For RI, we randomly change a certain percentage of features in the testing set to produce adversarial examples. When crafting adversarial samples, we follow [27, 59] to ensure the feature modifications are domain-mappable and can be repackaged to APKs. For feature-vector-based approaches like Drebin, we follow [27] by constraining modifications to vector bits that are 0, changing them to 1. This ensures that all required permissions and functions remain unchanged, keeping the app's functionality unaffected. For graph-based solutions, we introduce non-disruptive modifications to the graph structures that do not alter the app's functionality. The graph structure includes non-leaf nodes (user-defined functions) and leaf nodes (Android API). When adding a new edge, we select a non-leaf node in the graph and add a try block with a callee chosen from leaf nodes. Since leaf nodes do not call any other nodes, the added edge does not affect the app's functionality. To measure the robustness against adversarial attacks, we use two metrics from [59]: Adversarial Success Rate (ASR) and Adversarial Perturbation Ratio (APR). The detailed definitions can be found in the Appendix A.3. Both metrics range (0, 1). A higher ASR indicates increased vulnerability to adversarial attacks, while a higher APR suggests greater robustness against such attacks due to the model's ability to withstand perturbations.

By analyzing Table 11, we note that JSMA yields an average ASR of 98.8% across the evaluated approaches, indicating their vulnerability to such basic adversarial attacks, let alone more sophisticated ones [47]. The employed strategies are less effective on MamaDroid, HomDroid, and RAMDA. The reason for MamaDroid and HomDroid could be linked to their abstraction of APKs' graph structures. Interestingly, MamaDroid, HomDroid, and MalScan all utilize API calls and program graphs; the former two methods demonstrate greater resilience. The key difference lies in their handling of program graphs — MamaDroid abstracts these graphs using family and package names, and HomDroid employs social network triads for abstraction, in contrast to MalScan's direct use of the graphs. These abstractions make the malicious semantics encoded in graph structures more resistant to perturbation, enhancing the robustness of the detectors built on them. Additionally, RAMDA's resilience can be attributed to its customized Autoencoder, which learns compressed representations of benign apps, making it more defensive to adversarial examples. These findings suggest that abstracting feature representations or augmenting models' defensive capabilities could improve malware detectors' robustness to adversarial attacks.



(a) Time efficiency

(b) Memory efficiency

Fig. 8. The efficiency of extracting features from APKs in terms of time and memory.

Summary: ML-based malware detectors are inherently susceptible to real-world challenges such as malware evolution, code obfuscation, and adversarial attacks. Our findings suggest that detectors capable of capturing stable features and underlying malware semantics are more robust to these challenges. These observations highlight the need for future research to focus on identifying, modeling, and integrating semantically meaningful behavioral representations that remain invariant under transformation and evolution.

5.4 Efficiency

In this section, we scrutinize the efficiency of the selected methods across datasets from different years, focusing on the end-to-end efficiency of the entire detection pipeline, including feature extraction, transformation, and ML modeling. For feature extraction, we utilize average time for processing one APK and memory overhead during the extraction process. For feature transformation, we measure the time taken to retrieve and encode the required features from pre-extracted features. We normalize the time based on processing 20,000 apps per year. For ML modeling, we evaluate the time taken for model training and prediction. All experiments are performed on a server with 32-core CPUs operating at 2.10 GHz, 251 GB of physical memory, and two GPUs, each with 32 GB of memory.

Feature extraction. We first measure the performance overhead incurred during feature extraction from APKs. The extraction process supports multi-process execution to enable concurrent feature extraction; in our experiments, we set the number of processes to 16. We record the average time required to extract features from a single APK, as well as the average memory consumption. Memory usage is monitored across all processes at 30-second intervals, and the mean value is reported as the memory overhead.

Figure 8 illustrates the time and memory efficiency of feature extraction. We observe that both time and memory overheads increase over the years, a trend that can be attributed to the growing complexity and size of APKs, which require more resources for effective feature extraction. In general, time and memory consumption are positively correlated, with higher extraction times typically accompanied by higher memory usage. Despite this upward trend, the observed overheads remain acceptable for practical use of FrameDroid. For example, extracting features from an APK released in 2020 takes approximately 5.5 seconds, with an overall memory consumption of about 22.9 GB when using 16 concurrent processes — well within the capabilities of modern systems.

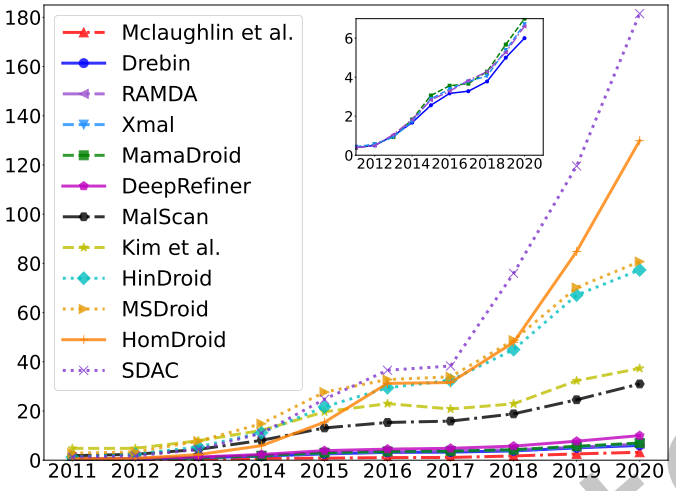


Fig. 9. The efficiency of feature transformation of the selected approaches. The x-axis shows the dataset year, and the y-axis indicates the utilized time in hours.

Table 12. The efficiency of selected approaches across datasets from different years, covering training and testing phases.

		Drebin	MamaDroid	Mclaughlin et al.	HinDroid	DeepRefiner	Kim et al.	MalScan	SDAC	HomDroid	Xmal	RAMDA	MSDroid
Training	**2011**	15.11s	1.90s	128m22s	2m19s	369m18s	98m47s	1m14s	84m47s	2.00s	1m15s	1m6s	20m19s
	2012	20.88s	1.94s	97m01s	2m18s	359m35s	92m47s	1m18s	89m05s	2.04s	1m25s	1m11s	37m54s
	2013	22.37s	2.12s	172m20s	2m13s	477m18s	115m40s	1m24s	115m35s	2.04s	50.36s	1m06s	40m11s
	2014	32.49s	2.46s	128m13s	2m40s	629m26s	104m03s	1m33s	132m55s	2.20s	2m20s	1m17s	42m07s
	2015	26.40s	2.60s	411m03s	2m44s	730m41s	115m27s	1m38s	250m07s	2.22s	2m10s	1m20s	42m52s
	2016	32.62s	2.99s	234m19s	2m51s	867m25s	135m50s	1m42s	470m38s	2.26s	1m05s	1m19s	142m23s
	2017	22.36s	2.75s	192m41s	2m28s	997m04s	145m24s	1m37s	296m05s	2.22s	1m54s	1m20s	47m22s
	2018	19.97s	2.91s	227m48s	2m36s	933m12s	128m32s	1m34s	716m43s	2.21s	1m19s	1m21s	155m50s
	2019	26.83s	3.03s	514m18s	2m34s	968m50s	185m43s	1m32s	918m09s	2.22s	1m34s	1m18s	130m41s
	2020	22.87s	2.69s	771m44s	2m22s	1064m30s	148m08s	1m29s	867m55s	2.17s	2m09s	1m12s	93m08s
Testing	**2011**	0.98s	0.06s	16.03s	29.82s	42.04s	0.62s	18.02s	54.43s	1.20s	0.32s	0.07s	4.33s
	2012	1.36s	0.06s	16.94s	30.40s	45.50s	0.89s	22.23s	53.97s	1.30s	0.32s	0.07s	6.15s
	2013	1.48s	0.06s	16.25s	31.53s	50.36s	1.10s	19.27s	1m21s	1.22s	0.34s	0.07s	7.14s
	2014	1.53s	0.06s	24.69s	33.10s	59.22s	1.11s	24.83s	1m51s	1.36s	0.32s	0.09s	11.29s
	2015	1.73s	0.06s	28.63s	34.95s	1m12s	0.97s	23.59s	2m54s	1.41s	0.39s	0.07s	10.92s
	2016	2.16s	0.06s	40.04s	36.50s	1m23s	1.19s	23.92s	3m37s	1.54s	0.33s	0.08s	18.72s
	2017	1.46s	0.06s	34.05s	35.25s	1m12s	0.92s	26.19s	4m16s	1.50s	0.31s	0.09s	17.96s
	2018	1.28s	0.06s	36.39s	35.17s	1m18s	0.89s	22.67s	6m21s	1.28s	0.32s	0.09s	17.96s
	2019	1.74s	0.06s	57.58s	34.93s	1m33s	1.18s	24.43s	8m55s	1.45s	0.33s	0.09s	21.14s
	2020	1.40s	0.06s	1m12s	33.90s	1m49s	0.91s	23.62s	8m27s	1.52s	0.41s	0.07s	19.54s

Feature transformation. With pre-extracted features available, we further evaluate the time efficiency of the selected methods in retrieving and encoding their required features. All feature transformation procedures are executed using 16 concurrent processes to ensure consistency across methods.

Figure 9 presents the time overhead incurred by different approaches during feature transformation. We observe a clear increasing trend over time, with more recent APKs requiring longer processing times. This trend is consistent with the rapid growth in both the size and structural complexity of Android applications, which imposes higher computational costs during feature encoding. These results highlight the importance of designing scalable and efficient feature encoding strategies to keep pace with the evolution of Android ecosystems. Notably, substantial variance in time consumption is observed across different detection approaches. This variation aligns with our expectations, as different methods rely on distinct feature types and encoding mechanisms with varying computational complexity. For example, SDAC exhibits the highest time overhead due to its need to recursively traverse the program graph to generate API call sequences, an operation that scales poorly with increasing code complexity. In contrast, approaches such as Xmal and RAMDA are considerably more time-efficient, as they rely on a single traversal of the program graph to extract API calls, resulting in significantly lower processing overhead.

Overall, these findings demonstrate that the choice of feature representation and encoding strategy has a substantial impact on time efficiency, underscoring a key trade-off between expressive feature modeling and computational scalability in ML-based Android malware detection.

ML modeling. For each yearly dataset, we split the samples into training, validation, and testing sets using an 8:1:1 ratio. We then measure the time required by the selected approaches for both model training and testing. For deep learning (DL)-based methods, we adopt an early-stopping strategy during training to prevent overfitting and ensure fair comparison.

Table 12 reports the training and testing time of the evaluated approaches. From a cross-method perspective, we observe that most DL-based techniques incur substantially higher training costs than traditional machine learning (TML) methods. Moreover, training time is strongly correlated with model complexity, with deeper or more expressive architectures requiring longer training durations. From a longitudinal perspective, training time consistently increases over the years, reflecting the growing complexity of Android applications and the expanding dimensionality of extracted features. When considered jointly with detection effectiveness (Section 5.2), these results reveal that higher resource consumption does not necessarily translate into superior detection performance. This observation highlights a fundamental trade-off between effectiveness and efficiency in ML-based Android malware detection. Designing practical detectors therefore requires careful consideration of this balance, where incorporating semantically meaningful features plays a critical role in achieving strong performance without excessive computational overhead. During the testing phase, we observe trends similar to those in training, with prediction time increasing modestly over time. Nevertheless, most approaches are able to complete inference within seconds when evaluating approximately 2,000 apps, indicating that they remain suitable for near real-time malware detection in practical deployment scenarios.

Summary: The basic costs of feature extraction and malware prediction for ML-based detectors remain within acceptable limits for practical deployment. However, the primary efficiency bottleneck lies in feature transformation, where both what features are selected and how they are represented have a significant impact on encoding time. Notably, detector efficiency does not necessarily correlate with detection effectiveness, underscoring the importance of balancing effectiveness and efficiency. In particular, features should be well aligned with the capabilities of the chosen model to achieve an optimal trade-off. For example, when employing classifiers such as SVMs, extracting complex graph-based features may be unnecessary, as such models are not well-suited to fully exploit the graph structure. Future work should focus on principled feature-model co-design, enabling efficient yet semantically expressive detection pipelines.

Table 13. Test Set Performance Comparison: APKTool vs Androguard Features

Model	APKTool				Androguard			
	Acc.	Prec.	Rec.	F1	Acc.	Prec.	Rec.	F1
RandomForest	0.975	0.941	0.800	0.865	0.984	0.886	0.975	0.929
SVM	0.963	0.931	0.675	0.783	0.978	0.921	0.875	0.897
DecisionTree	0.960	0.786	0.825	0.805	0.984	0.905	0.950	0.927
KNN	0.960	0.900	0.675	0.771	0.973	0.941	0.800	0.865
MLP	0.973	0.939	0.775	0.849	0.981	0.884	0.950	0.916

5.5 Further Experimental Validation

5.5.1 ***Influence of Reverse Engineering Tools.*** As discussed in Section 4.3, different reverse engineering tools may extract divergent features from the same APK, potentially affecting the performance of ML-based malware detectors. To systematically investigate this effect, we randomly select 4,000 apps from the additional dataset shown in Figure 4, maintaining a malware ratio of 10%. We then extract features using two widely adopted reverse engineering tools: Androguard [1] and APKTool [3]. In this experiment, we focus on two commonly used feature types—API calls and permissions. We split the dataset into training, validation, and testing sets, and train five representative ML models — Random Forest, SVM, Decision Tree, KNN, and MLP — using features extracted by each tool independently. Each trained model is evaluated on its corresponding test set, and the results are compared to assess the impact of tool choice on detection performance.

Our analysis reveals that the features extracted by the two tools are not always consistent. In some cases, certain API calls or permissions are identified by one tool but missed by the other. For instance, for the APK `00002EA4***0B6FB1`, Androguard extracts 15 additional API calls, such as `getRunningAppProcesses` and `getRunningTasks`, that are not detected by APKTool. Such discrepancies are expected, as different tools employ distinct parsing strategies and static analysis techniques, leading to variations in feature coverage. We further evaluate the performance of models trained on features extracted by each tool, with the results summarized in Table 13. The results show noticeable performance differences across models depending on the tool used for feature extraction, demonstrating that the choice of reverse engineering tool can significantly influence experimental outcomes This finding underscores the importance of standardizing the feature extraction toolchain when conducting comparative evaluations. Accordingly, in this work, we use the same tool for all approaches to ensure fair and reproducible comparisons. Given its consistently stronger performance across most settings, we select Androguard as the default feature extraction tool in FrameDroid.

5.5.2 ***Detection Effectiveness.*** As described in Section 5.1, we use the primary dataset spanning 2011-2020 as the main benchmark for evaluating the effectiveness of the selected representative approaches, providing a comprehensive view of the state of ML-based Android malware detection. To further validate the generalizability of our findings and examine whether they remain valid on more recent data, we conduct an additional evaluation using a supplementary dataset collected between 2021 and 2024, as illustrated in Figure 4. For consistency, we split the dataset into training, validation, and testing sets using an `8:1:1` ratio, while maintaining a goodware-to-malware ratio of 9:1, following the same experimental setup as in Section 5.2.

Table 14 reports the F1-score and accuracy of the selected approaches on the additional datasets. We observe trends consistent with those obtained from the primary dataset: under realistic settings, Hindroid, Kim et al., and Homdroid consistently outperform other methods and achieve comparable F1-score and accuracy. These results indicate that traditional machine learning (TML)-based and deep learning (DL)-based approaches continue to exhibit similar effectiveness on recent data, further supporting our earlier finding that DL-based methods may

Table 14. Average F1-Score and Accuracy for different methods on the additional datasets from 2021 to 2024.

Method	Drebin	Mamadroid	Mclaughin et al.	Hindroid	DeepRefiner	Kim et al.
F1-Score	0.857	0.716	0.717	0.900	0.707	0.879
Accuracy	0.973	0.744	0.744	0.981	0.742	0.977
Method	Malscan	SDAC	Homdroid	Xmal	RAMDA	MsDroid
F1-Score	0.716	0.735	0.888	0.670	0.723	0.823
Accuracy	0.743	0.747	0.979	0.734	0.932	0.958

require a larger volume of malware samples to effectively distill malicious semantics and surpass TML-based approaches.

This experiment serves as a lightweight yet fundamental validation that the selected representative approaches—and our key observations—remain effective on more recent datasets. Other aspects of detector behavior, such as robustness and efficiency, are less sensitive to dataset time periods and therefore are not re-evaluated on the additional dataset. Further discussion on the influence of dataset temporal characteristics is provided in Section 7.

6 Findings and Recommendations

We now draw from our findings to discuss the current state of ML-based Android malware detection and put forth recommendations to guide future research in this area.

Findings. We summarize our key findings as follows:

(1) Current ML-based Android malware detectors still face open challenges. While ML models have been evidently effective in detecting malware [48, 56, 61, 65], their effectiveness is still far from satisfactory when faced with challenging scenarios such as limited data size, rapid malware evolution, and adversarial attacks.

(2) Selecting features relevant to malware semantics is crucial for effective detection. A wide range of APK features, such as permissions and intents, have been leveraged for malware detection [17, 56, 102]. These features serve to profile app behavior and play a critical role in determining detector effectiveness. Our findings suggest that heuristic feature selection can effectively reduce noise in the feature space, allowing models to more readily identify malicious patterns — particularly in data-scarce settings. However, we also observe that the semantics captured by different feature types are not always complementary, and principled strategies for combining them remain underexplored. Consequently, naively incorporating additional features does not necessarily yield improved detection performance.

(3) More complex models are not a silver bullet in designing malware detectors. The literature reflects the trend towards employing more powerful ML models to detect malware [17, 102]. Our analysis reveals that DL-based methods appear to be more resilient than TML-based approaches in adapting to malware evolution. However, we also find that DL-based approaches are less effective than TML-based methods when the malware-to-goodware ratio is low. This suggests that utilizing more complex models is not a universal solution for malware detection. Instead, the choice of models should take into account the richness of the semantic information derived from the features. One more practical observation is given feature sets including discrete features (*i.e.,* permissions and API calls), ensembling models and DL models with embedding layers tend to outperform other typical models.

(4) Both feature abstraction and models' defensive mechanism contribute to detectors' robustness. Our analysis indicates that feature abstraction can help ML models capture more robust malicious patterns [22]. For example, MamaDroid [65] abstracts program graphs at the family and package levels, which enhances its robustness against adversarial attacks. In addition, strengthening the defensive capabilities of ML models can further improve

detector reliability and stability [73]. As an illustration, RAMDA [61] employs an autoencoder to reconstruct input features, learning compact and robust app representations that mitigate the impact of adversarial perturbations.
(5) Detection effectiveness does not positively correlate with efficiency. Through a combined analysis of effectiveness and efficiency, we observe that effectiveness and efficiency do not always positively correlate. A detector demanding more resources does not necessarily deliver enhanced results. For example, Drebin [17] can achieve competitive detection performance with relatively low resource consumption.

Recommendations. Our findings reveal that the effectiveness of ML-based Android malware detectors is closely tied to the quantity and quality of malware semantics extracted from APK features, which describe the malicious behaviors of apps. To design more effective and robust detectors, we recommend the following strategies:
(1) Quantify malware semantics. Rather than indiscriminately incorporating additional features to enhance detection performance, a more principled approach is to design metrics that explicitly evaluate both the quantity and quality of malware semantics captured by features, as well as the degree of redundancy among them. Such metrics could measure properties such as semantic coverage, discriminative power, and feature overlap across different behavior representations. By systematically quantifying these aspects, researchers can better identify features that are not only informative but also complementary, enabling more effective feature selection and combination strategies. Ultimately, this direction can lead to malware detectors that achieve stronger performance with fewer, semantically richer features, while also improving efficiency and robustness in real-world deployment scenarios.
(2) Investigate feature combination and abstraction. Given the diverse features available to describe app behaviors, a promising research direction is to explore effective strategies for combining these features to amplify their collective ability to capture malware semantics. Additionally, feature abstraction has demonstrated potential in enhancing the robustness of detectors. Future work could focus on developing techniques to abstract features in ways that improve detectors' resilience to real-world challenges while maintaining or even enhancing detection performance.
(3) Mine invariant malware semantics. Noticeable challenges in ML-based Android malware detection include vulnerability to malware evolution and susceptibility to adversarial attacks, both of which can significantly degrade detection performance in real-world deployments. To mitigate these challenges, future research should focus on identifying and modeling invariant malware semantics—high-level behavioral characteristics that remain stable across malware variants, transformations, and obfuscation techniques. Such semantics may capture fundamental malicious intents, such as unauthorized data access or covert communication patterns, rather than surface-level syntactic features. By grounding detection models in these invariant semantics, it becomes possible to build detectors that are more resilient to code evolution, packing, and adversarial manipulation.
(4) Align model complexity with malware semantics. Once the relevant features and their roles in capturing malware semantics are identified, an equally critical step is selecting models that can effectively leverage this semantic information. Model complexity should be carefully aligned with the richness and structure of the extracted semantics to achieve an optimal balance between detection effectiveness and computational efficiency. Overly complex models may fail to provide additional benefits when the input features lack sufficient semantic depth, while overly simplistic models may be incapable of exploiting rich behavioral representations. Future detector design should therefore emphasize feature-model co-design, ensuring that model expressiveness is commensurate with the semantic information available, particularly in resource-constrained or real-world deployment scenarios.

7 Discussion

Systematic investigation. To better understand the efforts dedicated to Android malware detection, we conduct a systematic literature review since 2011. Aiming for the inclusion of a broad range of papers, we follow the search strategies used in [63, 64]. Specifically, we perform searches in key digital libraries, such as the ACM

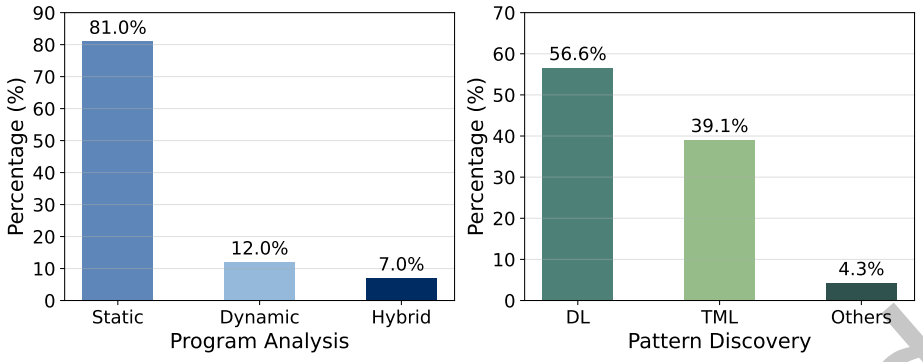


Fig. 10. The overall distribution of investigated approaches in Android malware detection.

Digital Library and IEEE Xplore, using specific keywords, including `android malware detection`, `android analysis`, and `android malware`. Then, a careful screening of titles and introductions is followed to selectively exclude studies unrelated to our research topic. This meticulous process ultimately leads to the identification of 258 related papers.

We subsequently categorize these papers based on the program analysis and pattern discovery techniques they employ. Figure 10 shows the distribution of these techniques. Our analysis reveals that *static analysis* is the predominant technique utilized to extract features from apps, followed by *dynamic* and *hybrid analysis*. For pattern discovery, *machine learning*, including both traditional machine learning (TML) and deep learning (DL) models, are extensively used to identify malicious patterns. Particularly notable is the significant increase in the utilization of static feature extraction and ML-based techniques in recent years. This evolving trend highlights the importance of our study, seeking to provide an in-depth understanding of the contemporary landscape in ML-based Android malware detection.

Dataset considerations. Including mobile apps from multiple markets is essential for improving the representativeness of an Android malware dataset. To incorporate diverse app sources, we select AndroZoo [13] as the primary data source, following common practice in prior studies [28, 77]. AndroZoo aggregates apps collected from multiple distribution channels, including Google Play as well as third-party markets such as Anzhi and AppChina, enabling the construction of a dataset that spans different app ecosystems rather than relying on a single market.

However, we acknowledge certain limitations of using AndroZoo as the primary malware source. Although AndroZoo integrates samples from various sources (e.g., VirusShare, AppChina, and Anzhi), a substantial portion of its apps originate from Google Play. As a result, some malware samples in AndroZoo may have previously passed Google Play's vetting process at some point in their lifecycle, which may introduce a potential bias. Specifically, such samples might not fully capture the characteristics of in-the-wild malware ecosystems, particularly those distributed through less regulated or underground channels. Beyond this, AndroZoo also does not cover all existing Android markets. For example, some prominent regional app stores—such as Huawei AppGallery and other OEM-operated markets—are not included. Incorporating apps from these additional sources, as well as alternative malware feeds, could further enhance dataset comprehensiveness and better reflect the diversity of the Android threat landscape.

Nevertheless, during dataset construction, we explicitly ensured that our samples include applications originating from multiple markets within AndroZoo, including Google Play, VirusShare, AppChina, Anzhi, and VirusTotal, rather than being dominated by a single source. Consequently, our analysis and findings are grounded in behaviors observed across diverse app markets, which mitigates the risk that our conclusions are driven by

market-specific biases. Moreover, different app markets adopt security policies and vetting mechanisms, which can affect both the prevalence and behavioral characteristics of malware. Although these vetting strategies often share common mechanisms — such as signature-based scanning and heuristic analysis — prior work has shown that they are inherently reactive and subject to non-negligible detection delays [43, 112]. To account for this, we adopt a long time span when constructing our dataset, allowing malicious apps to be labeled post hoc and increasing the likelihood that the collected malware samples are representative and behaviorally diverse. While acknowledging the dataset's limitations, we believe that our primary dataset is sufficiently diverse and representative to support a meaningful and realistic evaluation of ML-based Android malware detectors.

We assess the performance of the selected Android malware detectors using two datasets collected over different time periods: a primary dataset spanning 2010-2020 and an additional dataset from 2021-2024. The additional dataset is used to evaluate effectiveness, allowing us to examine whether detection performance trends remain consistent across temporally distinct Android ecosystems.

For the effectiveness evaluation, we observe that detection trends are consistent across both datasets, with similar performance rankings among the evaluated approaches. For example, HinDroid, Kim et al., and HomDroid achieve top performance in both time periods. This consistency indicates that our key findings regarding detection effectiveness are robust and generalizable across different time periods. We further examine performance variations across datasets from different time periods in Section 5.2 (*Detector stableness of different time period dataset*). The results show that the effectiveness of these detectors remains relatively stable across the two periods, with only minor fluctuations in performance metrics. This observation further supports the conclusion that our findings derived from the primary dataset are applicable to more recent Android ecosystems as well. For the robustness evaluation, the results are conceptually insensitive to the specific time period of the dataset. In malware evolution experiments, the key requirement is that training samples precede testing samples temporally. For obfuscation and adversarial evaluations, robustness is assessed by applying controlled transformations to testing apps, which does not depend on the absolute collection years of the dataset. For the efficiency evaluation, we measure performance using the primary dataset while explicitly accounting for temporal changes in app characteristics. The decade-long span of this dataset is sufficient to capture substantial changes in app complexity and scale, enabling a meaningful assessment of efficiency trends.

Overall, while the reliance on AndroZoo introduces inherent limitations in fully capturing the in-the-wild malware landscape, our dataset provides a large-scale, systematically curated, and widely adopted benchmark. Combined with cross-temporal evaluation and complementary robustness and efficiency analyses, it supports a meaningful and reproducible assessment of ML-based Android malware detectors.

App packing and its impact. To protect applications from reverse engineering and tampering, many developers employ app packing techniques that obfuscate the original code structure [31, 32]. Such techniques can substantially degrade the effectiveness of static analysis-based malware detectors, as they hinder the extraction of meaningful features from packed apps. In this study, to systematically examine the state of ML-based Android malware detection, we focus primarily on unpacked apps, which is consistent with the scope of most prior work [28, 97]. During dataset construction, we identify an app as packed if it employs any known packing tools (*e.g.,* Qihoo 360) and exclude these apps to ensure reliable static feature extraction. We acknowledge that packing techniques can increase static feature extraction failure rates, negatively impacting the performance of static analysis-based detectors. To mitigate this limitation in practical deployments, future detectors may benefit from integrating static and dynamic analysis techniques, enabling more robust feature extraction in the presence of packing and obfuscation. In this work, our objective is to evaluate and understand the performance characteristics of ML-based Android malware detectors under controlled and comparable conditions. A systematic investigation of app packing techniques and their impact on malware detection effectiveness is an important and complementary direction, which we leave for future work.

Potential of large language models in Android malware analysis. Large Language Models (LLMs) have demonstrated remarkable capabilities in natural language processing, code understanding, and reasoning tasks. Building upon the findings and insights from our study, we believe that LLMs hold substantial promise in the domain of Android malware analysis, particularly in the following areas: (1) Feature space exploration: LLMs can assist in selecting and combining discriminative features to enhance the performance of malware detectors. By analyzing the impact of different feature sets on app behavior description, LLMs can guide the construction of more effective and concise feature representations. (2) Semantic understanding of malicious behavior: LLMs can mine high-level malicious semantics directly from code, improving the interpretability of detection systems and offering deeper insight into the behavioral mechanisms of malware.

For example, LLMs can analyze multiple features that describe the same malicious behavior, evaluate their relative impact on detection performance, and identify the most informative feature combinations for comprehensive behavior modeling. Moreover, LLMs can summarize the semantics of groups of functions within an application to uncover specific malicious operations—such as sensitive information collection, data exfiltration, or command-and-control communication. These semantic summaries can then be leveraged to enhance app characterization and improve the detection of potentially malicious behaviors.

Threats to validity. There are two main threats to the validity of our study. First, our research mainly focuses on investigating general ML-based Android malware detectors. That is, we do not include the methods designed to solve a particular challenge like malware evolution. Specifically, there have been several attempts [28, 72, 100, 108] starting to mitigate the challenges we have identified. For instance, the recent APIGraph [108] identifies semantically similar API calls to enhance detectors' robustness against malware evolution. Integrating this strategy with popular detectors like Drebin and MamaDroid leads to 5% - 10% detection improvements over a one-year malware evolution. Nevertheless, our findings still hold, as these improvements are insufficient compared to the reduction of around 30% observed in our study. It is our aspiration that this study can motivate more researchers to focus on these challenges and develop effective solutions to mitigate them.

Second, inconsistencies might exist between our evaluation and the reported ones due to different settings, such as datasets, metrics, and toolchains. To mitigate experimental biases and provide a fair comparison, we design a general-purpose framework. Specifically, we adopt standardized techniques across all tasks, including feature extraction and ML modeling. We also incorporate many evaluation scenarios, such as training data sizes, malware evolution, and efficiency, to ensure a comprehensive measurement using our crafted dataset. It is our hope that the framework can facilitate future work in ML-based Android malware detection.

8 Conclusion

This paper performs the most extensive systematic study of the ML-based Android malware detection literature with empirical and quantitative analysis. We identify challenges (*i.e.,* unfair comparisons, unrealistic evaluations, and unclear computational costs) that hinder the systematization in this field. In response, we design a general-purpose framework for developing ML-based detection approaches and evaluating their effectiveness, robustness, and efficiency. By experimentally comparing 12 representative approaches, our study paints a holistic view of the state of ML-based Android malware detection and puts forth recommendations to guide future research. For instance, we find existing detectors are still vulnerable to malware evolution and adversarial attacks and in future work, we should focus on incorporating more *malware semantics* to design more practical detectors.

Acknowledgments

We thank Bonan Ruan, Jiawei Li, and Chuqi Zhang for their assistance and the anonymous reviewers for their valuable comments. This research is partially supported by the National Research Foundation, Singapore, through the National Cybersecurity R&D Lab at the National University of Singapore under its National Cybersecurity

R&D Programme (Award No. NCR25-NCL P3-0001) and by UK EPSRC Grant no. EP/X015971/2. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors(s) and do not reflect the views of National Research Foundation, Singapore, National Cybersecurity R&D Lab at the National University of Singapore, and EPSRC.

References

[1] [n. d.]. Androguard. https://github.com/androguard/.
[2] [n. d.]. Angr. https://angr.io/.
[3] [n. d.]. APKTool. https://ibotpeaches.github.io/Apktool/.
[4] [n. d.]. BackSmali. https://github.com/JesusFreke/smali.
[5] [n. d.]. How Many Apps In Google Play Store? https://www.bankmycell.com/blog/number-of-google-play-store-apps.
[6] [n. d.]. IDA Pro. https://hex-rays.com/ida-pro/.
[7] [n. d.]. Kharon project. https://cidre.gitlabpages.inria.fr/malware/malware-website/dataset/malware_DroidKungFu1.html.
[8] [n. d.]. LibRadar. https://github.com/pkumza/LibRadar.
[9] [n. d.]. PyTorch. https://pytorch.org/.
[10] [n. d.]. VirusShare. https://virusshare.com/.
[11] [n. d.]. VirusTotal. https://www.virustotal.com.
[12] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. Droidapiminer: Mining api-level features for robust malware detection in android. In *International ICST Conference, SecureComm*.
[13] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *MSR*.
[14] Muhammad Amin, Babar Shah, Aizaz Sharif, Tamleek Ali, Ki-Il Kim, and Sajid Anwar. 2022. Android malware detection through generative adversarial networks. *Emerging Telecommunications Technologies* (2022).
[15] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. 2020. Obfuscapk: An open-source black-box obfuscation tool for Android apps. *SoftwareX* (2020).
[16] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and don'ts of machine learning in computer security. In *Security*.
[17] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket.. In *NDSS*.
[18] Anish Athalye, Nicholas Carlini, and David Wagner. 2018. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *ICML*.
[19] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: analyzing the android permission specification. In *CCS*.
[20] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On demystifying the android application framework:{Re-Visiting} android permission specification analysis. In *Security*.
[21] Federico Barbero, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. 2022. Transcending transcend: Revisiting malware classification in the presence of concept drift. In *SP*.
[22] Arjun Nitin Bhagoji, Daniel Cullina, Chawin Sitawarin, and Prateek Mittal. 2018. Enhancing robustness of machine learning systems via data transformations. In *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*. IEEE.
[23] Minghui Cai, Yuan Jiang, Cuiying Gao, Heng Li, and Wei Yuan. 2021. Learning features from enhanced function call graphs for Android malware detection. *Neurocomputing* (2021).
[24] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *S&P*.
[25] Fabrício Ceschin, Marcus Botacin, Albert Bifet, Bernhard Pfahringer, Luiz S Oliveira, Heitor Murilo Gomes, and André Grégio. 2020. Machine learning (in) security: A stream of problems. *Digital Threats: Research and Practice* (2020).
[26] Simin Chen, Soroush Bateni, Sampath Grandhi, Xiaodi Li, Cong Liu, and Wei Yang. 2020. DENAS: automated rule generation by knowledge extraction from neural networks. In *ESEC/FSE*.
[27] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. 2019. Android HIV: A study of repackaging malware for evading machine-learning detection. *TIFS* (2019).
[28] Yizheng Chen, Zhoujie Ding, and David Wagner. 2023. Continuous Learning for Android Malware Detection. *arXiv preprint arXiv:2302.04332* (2023).
[29] Nadia Daoudi, Jordan Samhi, Abdoul Kader Kabore, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2021. Dexray: a simple, yet effective deep learning approach to android malware detection based on image representation of bytecode. In *DMLSD*.
[30] Yuxin Ding, Xiao Zhang, Jieke Hu, and Wenting Xu. 2023. Android malware detection method based on bytecode image. *Journal of Ambient Intelligence and Humanized Computing* (2023).

[31] Zikan Dong, Hongxuan Liu, Liu Wang, Xiapu Luo, Yao Guo, Guoai Xu, Xusheng Xiao, and Haoyu Wang. 2022. What did you pack in my app? a systematic analysis of commercial Android packers. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1430–1440.
[32] Yue Duan, Mu Zhang, Abhishek Vasist BHASKAR, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang. 2018. Things you may not know about android (un) packers: A systematic study based on whole-system emulation. (2018).
[33] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. On lightweight mobile phone application certification. In *CCS*.
[34] Yujie Fan, Mingxuan Ju, Shifu Hou, Yanfang Ye, Wenqiang Wan, Kui Wang, Yinming Mei, and Qi Xiong. 2021. Heterogeneous temporal graph transformer: An intelligent system for evolving android malware detection. In *KDD*.
[35] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. 2014. Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys tutorials* (2014).
[36] Ruitao Feng, Sen Chen, Xiaofei Xie, Lei Ma, Guozhu Meng, Yang Liu, and Shang-Wei Lin. 2019. Mobidroid: A performance-sensitive malware detection system on mobile platform. In *ICECCS*. IEEE.
[37] Ruitao Feng, Sen Chen, Xiaofei Xie, Guozhu Meng, Shang-Wei Lin, and Yang Liu. 2020. A performance-sensitive malware detection system using deep learning on mobile devices. *TIFS* (2020).
[38] Cuiying Gao, Gaozhun Huang, Heng Li, Bang Wu, Yueming Wu, and Wei Yuan. 2024. A Comprehensive Study of Learning-based Android Malware Detectors under Challenging Environments. In *ICSE*.
[39] Han Gao, Shaoyin Cheng, and Weiming Zhang. 2021. GDroid: Android malware detection and classification with graph convolutional network. *Computers & Security* (2021).
[40] Joshua Garcia, Mahmoud Hammad, and Sam Malek. 2018. Lightweight, obfuscation-resilient detection and family identification of android malware. *TOSEM* (2018).
[41] Ross Girshick. 2015. Fast r-cnn. In *ICCV*.
[42] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *NIPS*.
[43] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 281–294.
[44] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2017. Adversarial examples for malware detection. In *ESORICS*.
[45] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask r-cnn. In *CVPR*.
[46] Ke He and Dong-Seong Kim. 2019. Malware detection with malware images using deep learning techniques. In *TrustCom*.
[47] Ping He, Yifan Xia, Xuhong Zhang, and Shouling Ji. 2023. Efficient Query-Based Attack against ML-Based Android Malware Detection under Zero Knowledge Setting. In *CCS*.
[48] Yiling He, Yiping Liu, Lei Wu, Ziqi Yang, Kui Ren, and Zhan Qin. 2022. Msdroid: Identifying malicious snippets for android malware detection. *TDSC* (2022).
[49] Geoffrey Hinton. 2009. Deep belief networks. *Scholarpedia* (2009).
[50] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. 2017. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In *KDD*.
[51] TonTon Hsien-De Huang and Hung-Yu Kao. 2018. R2-d2: Color-inspired convolutional neural network (cnn)-based android malware detections. In *Big Data*. IEEE.
[52] Na Huang, Ming Xu, Ning Zheng, Tong Qiao, and Kim-Kwang Raymond Choo. 2019. Deep android malware classification with API-based feature graph. In *TrustCom/BigDataSE*.
[53] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. 2017. Transcend: Detecting concept drift in malware classification models. In *Security*.
[54] ElMouatez Billah Karbab and Mourad Debbabi. 2021. Petadroid: adaptive android malware detection using deep learning. In *DIMVA*.
[55] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. 2018. MalDozer: Automatic framework for android malware detection using deep learning. *Digital Investigation* (2018).
[56] TaeGuen Kim, BooJoong Kang, Mina Rho, Sakir Sezer, and Eul Gyu Im. 2018. A multimodal deep learning method for android malware detection using various features. *TIFS* (2018).
[57] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).
[58] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *ICML*.
[59] Heng Li, Zhang Cheng, Bang Wu, Liheng Yuan, Cuiying Gao, Wei Yuan, and Xiapu Luo. 2023. Black-box Adversarial Example Attack towards FCG Based Android Malware Detection under Incomplete Feature Information. In *Security*.
[60] Heng Li, ShiYao Zhou, Wei Yuan, Jiahuan Li, and Henry Leung. 2019. Adversarial-example attacks toward android malware detection system. *IEEE Systems Journal* (2019).

[61] Heng Li, Shiyao Zhou, Wei Yuan, Xiapu Luo, Cuiying Gao, and Shuiyan Chen. 2021. Robust android malware detection against adversarial example attacks. In *WWW*.
[62] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. Palmtree: Learning an assembly language model for instruction embedding. In *CCS*.
[63] Kaijun Liu, Shengwei Xu, Guoai Xu, Miao Zhang, Dawei Sun, and Haifeng Liu. 2020. A review of android malware detection approaches based on machine learning. *IEEE Access* (2020).
[64] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Deep learning for android malware defenses: a systematic literature review. *Comput. Surveys* (2022).
[65] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2017. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *NDSS*.
[66] Alejandro Martín, Félix Fuentes-Hurtado, Valery Naranjo, and David Camacho. 2017. Evolving deep neural networks architectures for android malware classification. In *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE.
[67] Niall McLaughlin, Jesus Martinez del Rincon, BooJoong Kang, Suleiman Yerima, Paul Miller, Sakir Sezer, Yeganeh Safaei, Erik Trickel, Ziming Zhao, Adam Doupé, et al. 2017. Deep android malware detection. In *CODASPY*.
[68] Larry R Medsker and LC Jain. 2001. Recurrent neural networks. *Design and Applications* (2001).
[69] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *NIPS*.
[70] Brad Miller, Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Rekha Bachwani, Riyaz Faizullabhoy, Ling Huang, Vaishaal Shankar, Tony Wu, George Yiu, et al. 2016. Reviewer integration and performance measurement for malware detection. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016*.
[71] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. 2017. graph2vec: Learning distributed representations of graphs. *arXiv:1707.05005* (2017).
[72] Annamalai Narayanan, Liu Yang, Lihui Chen, and Liu Jinliang. 2016. Adaptive and scalable android malware detection through online learning. In *IJCNN*. IEEE.
[73] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. 2016. Distillation as a defense to adversarial perturbations against deep neural networks. In *SP*.
[74] Naser Peiravian and Xingquan Zhu. 2013. Machine learning for android malware detection using permission and api calls. In *2013 IEEE 25th international conference on tools with artificial intelligence*. IEEE.
[75] Abdurrahman Pektaş and Tankut Acarman. 2020. Deep learning for effective Android malware detection using API call graph embeddings. *Soft Computing* (2020).
[76] Abdurrahman Pektaş and Tankut Acarman. 2020. Learning to detect Android malware via opcode sequences. *Neurocomputing* (2020).
[77] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, Lorenzo Cavallaro, et al. 2019. TESSERACT: Eliminating experimental bias in malware classification across space and time. In *Security*.
[78] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. 2020. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE.
[79] Junyang Qiu, Jun Zhang, Wei Luo, Lei Pan, Surya Nepal, and Yang Xiang. 2020. A survey of android malware detection with deep neural models. *ACM Computing Surveys (CSUR)* (2020).
[80] Alireza Souri and Rahil Hosseini. 2018. A state-of-the-art survey of malware detection approaches using data mining techniques. *Human-centric Computing and Information Sciences* 8, 1 (2018), 1–22.
[81] Xin Su, Dafang Zhang, Wenjia Li, and Kai Zhao. 2016. A deep learning approach to android malware feature learning and detection. In *Trustcom/BigDataSE*.
[82] Bo Sun, Tao Ban, Shun-Chieh Chang, Yeali S Sun, Takeshi Takahashi, and Daisuke Inoue. 2019. A scalable and accurate feature representation method for identifying malicious mobile applications. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*.
[83] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. 2017. Revisiting unreasonable effectiveness of data in deep learning era. In *CVPR*.
[84] Lichao Sun, Zhiqiang Li, Qiben Yan, Witawas Srisa-an, and Yu Pan. 2016. SigPID: significant permission identification for android malware detection. In *2016 11th international conference on malicious and unwanted software (MALWARE)*. IEEE.
[85] Mingshen Sun, Tao Wei, and John CS Lui. 2016. Taintart: A practical multi-level information-flow tracking system for android runtime. In *CCS*.
[86] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S Yu, and Tianyi Wu. 2011. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *Proceedings of the VLDB Endowment* (2011).
[87] Zhaonan Sun, Nawanol Ampornpunt, Manik Varma, and Svn Vishwanathan. 2010. Multiple kernel learning and the SMO algorithm. In *NIPS*.
[88] Fnu Suya, Jianfeng Chi, David Evans, and Yuan Tian. 2020. Hybrid batch attacks: Finding black-box adversarial examples with limited queries. In *Security*.

[89] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).
[90] R Vinayakumar, KP Soman, and Prabaharan Poornachandran. 2017. Deep android malware detection and classification. In *2017 International conference on advances in computing, communications and informatics (ICACCI)*. IEEE.
[91] Ji Wang, Qi Jing, Jianbo Gao, and Xuanwei Qiu. 2020. SEdroid: A robust Android malware detector using selective ensemble learning. In *2020 IEEE wireless communications and networking conference (WCNC)*. IEEE.
[92] Wei Wang, Mengxue Zhao, and Jigang Wang. 2019. Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. *Journal of Ambient Intelligence and Humanized Computing* (2019).
[93] Wikipedia contributors. [n. d.]. APK File Format. https://en.wikipedia.org/wiki/Apk_file_format.
[94] Bozhi Wu, Sen Chen, Cuiyun Gao, Lingling Fan, Yang Liu, Weiping Wen, and Michael R Lyu. 2021. Why an android app is classified as malware: Toward malware classification interpretation. *TOSEM* (2021).
[95] Songyang Wu, Pan Wang, Xun Li, and Yong Zhang. 2016. Effective detection of android malware based on the usage of data flow APIs and machine learning. *Information and software technology* (2016).
[96] Yueming Wu, Xiaodi Li, Deqing Zou, Wei Yang, Xin Zhang, and Hai Jin. 2019. Malscan: Fast market-wide mobile malware scanning by social-network centrality analysis. In *ASE*.
[97] Yueming Wu, Deqing Zou, Wei Yang, Xiang Li, and Hai Jin. 2021. HomDroid: detecting Android covert malware by social-network homophily analysis. In *ISSTA*.
[98] Xusheng Xiao and Shao Yang. 2019. An image-inspired and cnn-based android malware detection approach. In *ASE*.
[99] Jiayun Xu, Yingjiu Li, Robert H Deng, and Ke Xu. 2020. Sdac: A slow-aging solution for android malware detection using semantic distance based api clustering. *TDSC* (2020).
[100] Ke Xu, Yingjiu Li, Robert Deng, Kai Chen, and Jiayun Xu. 2019. Droidevolver: Self-evolving android malware detection system. In *EuroS&P*.
[101] Ke Xu, Yingjiu Li, and Robert H Deng. 2016. Iccdetector: Icc-based malware detection on android. *TIFS* (2016).
[102] Ke Xu, Yingjiu Li, Robert H Deng, and Kai Chen. 2018. Deeprefiner: Multi-layer android malware detection system applying deep neural networks. In *EuroS&P*.
[103] Jinpei Yan, Yong Qi, and Qifan Rao. 2018. LSTM-based hierarchical denoising network for Android malware detection. *Security and Communication Networks* (2018).
[104] Limin Yang, Wenbo Guo, Qingying Hao, Arridhana Ciptadi, Ali Ahmadzadeh, Xinyu Xing, and Gang Wang. 2021. {CADE}: Detecting and explaining concept drift samples for security applications. In *Security*.
[105] Yanfang Ye, Shifu Hou, Lingwei Chen, Jingwei Lei, Wenqiang Wan, Jiabin Wang, Qi Xiong, and Fudong Shao. 2019. Out-of-sample node representation learning for heterogeneous graph in real-time android malware detection. In *IJCAI*.
[106] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. 2014. Droid-sec: deep learning in android malware detection. In *SIGCOMM*.
[107] Peter Zegzhda, Dmitry Zegzhda, Evgeny Pavlenko, and Gleb Ignatev. 2018. Applying deep learning techniques for Android malware detection. In *Proceedings of the 11th International Conference on Security of Information and Networks*.
[108] Xiaohan Zhang, Yuan Zhang, Ming Zhong, Daizong Ding, Yinzhi Cao, Yukun Zhang, Mi Zhang, and Min Yang. 2020. Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In *CCS*.
[109] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *ESEC/FSE*.
[110] Kaifa Zhao, Hao Zhou, Yulin Zhu, Xian Zhan, Kai Zhou, Jianfeng Li, Le Yu, Wei Yuan, and Xiapu Luo. 2021. Structural attack against graph based android malware detection. In *CCS*.
[111] Yanjie Zhao, Li Li, Haoyu Wang, Haipeng Cai, Tegawendé F Bissyandé, Jacques Klein, and John Grundy. 2021. On the impact of sample duplication in machine-learning-based android malware detection. *TOSEM* (2021).
[112] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*. IEEE, 95–109.
[113] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets.. In *NDSS*.
[114] Dali Zhu, Yuchen Ma, Tong Xi, and Yiming Zhang. 2019. FSNet: android malware detection with only one feature. In *2019 IEEE Symposium on Computers and Communications (ISCC)*. IEEE.
[115] Dali Zhu, Tong Xi, Pengfei Jing, Di Wu, Qing Xia, and Yiming Zhang. 2019. A transparent and multimodal malware detection method for android apps. In *Proceedings of the 22nd international ACM conference on modeling, analysis and simulation of wireless and mobile systems*.
[116] Hui-Juan Zhu, Tong-Hai Jiang, Bo Ma, Zhu-Hong You, Wei-Lei Shi, and Li Cheng. 2018. HEMD: a highly efficient random forest-based malware detection framework for Android. *Neural Computing and Applications* (2018).

[117] Hui-Juan Zhu, Liang-Min Wang, Sheng Zhong, Yang Li, and Victor S Sheng. 2021. A hybrid deep network framework for android malware detection. *IEEE Transactions on Knowledge and Data Engineering* (2021).

Table 15. An overview of the 12 representative approaches we selected from major venues in security, software engineering, and machine learning. ● indicates that the APK file (*e.g.*, *AndroidManifest.xml* for Manifest) is taken as input in Android malware detection, while ○ is the opposite. ✔ and ✘ indicate whether feature engineering is handcrafted using domain knowledge or learned using representation learning. The effectiveness is based on the results reported in the original papers.

Selected Approach	Publication		Input from APK				Feature Engineering		Dataset		Effectiveness		
	Venue	Year	Manifest	Dex	Resource	Library	Handcrafted	Learned	Malware	Goodware	TPR	FPR	F1
Drebin[17]	NDSS	2014	●	●	○	○	✔	✘	5,560	123,453	94%	**1%**	87%
MamaDroid[65]	NDSS	2017	○	●	○	○	✔	✘	35,493	8,447	97%	2%	96%
Mclaughlin et al.[67]	CODASPY	2017	○	●	○	○	✘	✔	13,637	13,758	95%	**1%**	97%
HinDroid[50]	KDD	2017	○	●	○	○	✔	✘	1,216	1,118	**99%**	2%	**99%**
DeepRefiner[102]	EuroS&P	2018	●	●	●	○	✘	✔	62,915	47,525	98%	2%	98%
Kim et al.[56]	TIFS	2018	●	●	○	●	✔	✔	21,260	20,000	**99%**	**1%**	**99%**
MalScan[96]	ASE	2019	○	●	○	○	✔	✘	15,430	15,285	—	—	98%
SDAC[99]	TDSC	2020	○	●	○	○	✔	✔	34,497	35,645	98%	**1%**	**99%**
HomDroid[97]	ISSTA	2021	○	●	○	○	✔	✘	3,358	4,840	97%	4%	95%
Xmal[94]	TOSEM	2021	●	●	○	○	✔	✘	15,570	20,120	98%	2%	98%
RAMDA[61]	WWW	2021	●	●	○	○	✔	✘	21,621	36,862	93%	**1%**	90%
MSDroid[48]	TDSC	2022	○	●	○	○	✔	✘	30,210	51,580	97%	**1%**	97%

TPR, FPR, and F1 refer to true positive rate, false positive rate, and F1-score. As MalScan is only evaluated on F1, its TPR and FPR are not available.

A Appendix

This section contains additional data and information that can be of interest to the readers but that are not strictly necessary for understanding the main text.

A.1 Feature Database

To streamline the evaluation of different approaches, we establish a feature database to maintain commonly used features in Android malware detection. Below, we outline the main features incorporated into the database, categorizing them for ease of reference.

- *Manifest*: this category contains features sourced from the *AndroidManifest.xml* file, such as hardware components, permissions, intents, and app components.
- *DisassembledCode*: this category includes the disassembled data extracted from the DEX file, such as the opcode, operands, and code strings.
- *ProgramGraph*: this is mainly used to store the program graph of the APK file, including the nodes and edges.
- *SharedLibrary*: this category contains the information of the shared libraries used by the APK file.
- *Others*: this category includes other features that are not covered by the above categories.

Given the structure of the feature database, adding new features becomes straightforward, facilitating the adaptive evaluation of diverse approaches. For instance, we can easily implement an add-on feature extractor and store the derived features in the database, waiting for use by the preprocessor.

A.2 Reproduction of Selected Approaches

Table 15 offers a summary of the 12 representative approaches we have selected from leading publications in security [17, 48, 56, 65, 67, 99, 102], software engineering [94, 96, 97], and machine learning [50, 61]. Within this table, we outline the publication details, input from APK, feature engineering style, dataset statistics, and their original effectiveness. To better support subsequent research and foster replicability, we provide the hyperparameters of these approaches.

Drebin. We replicate Drebin [17] utilizing a linear SVM with $C = 1$, and set max number of iterations to 1000.
MamaDroid. MamaDroid [65] has two abstract mode *i.e.*, package and family. In our experiments, we chose the family mode, as it is more efficient in terms of time and memory. For the RF classifier, we employ the default hyper-parameters provided by scikit-learn.
Mclaughlin et al. Mclaughlin et al. [67] is implemented with a CNN with one single convolutional layer, followed by a max-pooling layer and a fully connected layer. The convolutional layer is configured with 32 filters and a kernel size of $225 * 7$. The fully connected layer has 16 neurons. For the evaluation process, a learning rate of 0.01 and a batch size of 32 are employed. Additionally, inputs that exceed a length of 600000 are truncated to 60000, and those that are less than 600000 are padded with zeros.
HinDroid. In implementing HinDroid [50], various meta-path combinations have been tested. AA^T yields remarkable performance in our experiments, and hence, is chosen as the meta-path. For multi-kernel learning, we utilize the $p - norm$ multi-kernel learning framework as released by [87], applying the default settings for the hyper-parameters.
DeepRefiner. As discussed in Section 4.2, DeepRefiner [102] has two detection layers, and the second layer shows strong detection capability. In our experiments, we replicate the second LSTM-based detection layer. The model configuration is as follows: an embedding size of 16 for word embeddings (bytecode instructions), a two-layer LSTM model with an input size of 16, and a hidden size of 64. Batch size is set as 32, and the learning rate is 0.001. The input sequence has a maximum length of $50,000$. Inputs exceeding this length are truncated, while shorter inputs are padded with zeros.
Kim et al. Kim et al. [56] initially use a 5 separated MLPs to process features from five various modalities, each with the same configuration. Subsequently, a new MLP is introduced to integrate the learned features from the previous MLPs. The initial five MLPs have layer configurations of 5000, 2500, and 1000 neurons. The last MLP is structured with layers containing 1000, 500, 100, and 10 neurons, respectively. A learning rate of 0.001 and a batch size of 32 are used during the training process.
MalScan. The algorithm [96] is replicated with a KNN classifier, with the number of neighbors set to 3.
SDAC. SDAC [99] initially employs Word2Vec to encode API calls into vectors. Following this, K-means clustering is applied, and then these cluster centers are used as anchors to encode features. In the classification phase, for simplicity, we use a linear SVM instead of multi-voting SVMs. Additionally, due to K-means's randomness, the algorithm is executed 5 times, and the average results are computed. The size of Word2Vec embedding is set as 10. The chosen number of clusters is 1000, and the SVM is configured using default parameters, allowing for 5000 iterations.
HomDroid. The approach [97] is executed using a KNN classifier, with the number of neighbors set to 1.
Xmal. Xmal [94] is implemented using a 3-layer MLP with a hidden size of 64. An attention layer is also incorporated, utilizing an MLP with a hidden size of 158. The learning rate is set as 0.001, and the batch size is 20.
RAMDA. RAMDA [61]'s architecture consists of two parts: autoencoder and classifier. The encoder and decoder each consist of a 3-layer MLP with a hidden size of 600. The classifier is a 4-layer MLP with a hidden size of 600. During the training process, the autoencoder is trained first, and then the classifier is trained. Configuration parameters are established with a learning rate of 0.001, a batch size of 64, and epochs set at 20. A pre-defined reconstruction loss of 30 is used. To constrain the loss, the positive weights are defined as $\lambda_1 = 10, \lambda_2 = 1, \lambda_3 = 10$.
MSDroid. MSDroid [48] is implemented using a 3-layer GNN with a hidden size of 512. Subsequent to this, a 2-layer fully connected network with a hidden size of 512 is used to classify the APKs. The model's training parameters are set with a learning rate of 0.01 and a batch size of 64.

Table 16. The AUT(TPR, N) and AUT(FRP, N) of the selected approaches over varying time decays. In this analysis, we examine the shifts during various malware evolution periods: 3, 6, 9, 12, 15, 18, 21, and 24 months.

Approach	AUT(TPR,0)	AUT(TPR,3)	AUT(TPR,6)	AUT(TPR,9)	AUT(TPR,12)	AUT(TPR,15)	AUT(TPR,18)	AUT(TPR,21)	AUT(TPR,24)
Drebin	0.803	0.722(-10.1%)	0.670(-16.5%)	0.601(-25.1%)	0.564(-29.8%)	0.530(-34.0%)	0.508(-36.7%)	0.484(-39.7%)	0.466(-41.9%)
MamaDroid	0.712	0.496(-30.3%)	0.429(-39.7%)	0.365(-48.7%)	0.322(-54.7%)	0.308(-56.7%)	0.282(-60.4%)	0.253(-64.5%)	0.233(-67.2%)
Mclaughlin et al.	0.724	0.583(-19.6%)	0.526(-27.3%)	0.461(-36.4%)	0.410(-43.4%)	0.378(-47.9%)	0.354(-51.2%)	0.327(-54.9%)	0.303(-58.1%)
HinDroid	0.831	0.786(-5.5%)	0.773(-7.0%)	0.729(-12.3%)	0.687(-17.4%)	0.684(-17.8%)	0.674(-19.0%)	0.662(-20.3%)	0.655(-21.2%)
DeepRefiner	0.774	0.590(-23.7%)	0.545(-29.6%)	0.478(-38.3%)	0.427(-44.8%)	0.413(-46.6%)	0.388(-49.9%)	0.362(-53.2%)	0.339(-56.2%)
Kim et al.	0.845	0.760(-10.0%)	0.745(-11.8%)	0.676(-20.0%)	0.619(-26.7%)	0.583(-31.0%)	0.557(-34.0%)	0.524(-38.0%)	0.49(-42.0%)
MalScan	0.800	0.685(-14.4%)	0.650(-18.8%)	0.584(-27.0%)	0.547(-31.6%)	0.519(-35.2%)	0.494(-38.3%)	0.465(-41.8%)	0.439(-45.1%)
SDAC	0.734	0.616(-16.1%)	0.554(-24.5%)	0.495(-32.6%)	0.455(-38.0%)	0.439(-40.2%)	0.416(-43.4%)	0.391(-46.8%)	0.369(-49.7%)
HomDroid	0.836	0.689(-17.5%)	0.651(-22.1%)	0.596(-28.7%)	0.546(-34.6%)	0.515(-38.4%)	0.486(-41.8%)	0.448(-46.4%)	0.422(-49.5%)
Xmal	0.836	0.741(-11.3%)	0.693(-17.1%)	0.617(-26.1%)	0.575(-31.2%)	0.550(-34.2%)	0.523(-37.4%)	0.492(-41.2%)	0.463(-44.6%)
RAMDA	0.876	0.814(-7.0%)	0.775(-11.6%)	0.733(-16.3%)	0.689(-21.3%)	0.673(-23.2%)	0.658(-24.8%)	0.637(-27.2%)	0.613(-30.1%)
MSDroid	0.835	0.755(-9.6%)	0.732(-12.3%)	0.673(-19.4%)	0.635(-24.0%)	0.632(-24.3%)	0.604(-27.6%)	0.569(-31.9%)	0.544(-34.8%)
Approach	**AUT(FPR,0)**	**AUT(FPR,3)**	**AUT(FPR,6)**	**AUT(FPR,9)**	**AUT(FPR,12)**	**AUT(FPR,15)**	**AUT(FPR,18)**	**AUT(FPR,21)**	**AUT(FPR,24)**
Drebin	0.014	0.02(+42.2%)	0.023(+68.2%)	0.024(+71.1%)	0.026(+85.6%)	0.029(+110.1%)	0.030(+115.2%)	0.031(+126.7%)	0.033(+141.2%)
MamaDroid	0.009	0.010(+17.9%)	0.011(+28.4%)	0.010(+20.2%)	0.011(+31.9%)	0.013(+50.5%)	0.013(+54.0%)	0.014(+62.2%)	0.014(+59.9%)
Mclaughlin et al.	0.013	0.010(-19.1%)	0.012(-7.5%)	0.013(+1.9%)	0.015(+14.3%)	0.016(+23.6%)	0.016(+24.4%)	0.016(+22.1%)	0.016(+20.5%)
HinDroid	0.016	0.262(+1491.6%)	0.266(+1516.6%)	0.271(+1548.8%)	0.279(+1599.3%)	0.319(+1839.1%)	0.321(+1853.7%)	0.324(+1873.8%)	0.327(+1890.3%)
DeepRefiner	0.017	0.019(+9.3%)	0.018(+5.9%)	0.019(+7.6%)	0.021(+20.3%)	0.022(+27.2%)	0.022(+28.9%)	0.023(+31.2%)	0.024(+38.7%)
Kim et al.	0.014	0.017(+19.7%)	0.018(+29.8%)	0.018(+31.9%)	0.02(+43.5%)	0.024(+75.9%)	0.026(+84.6%)	0.027(+91.8%)	0.027(+95.4%)
MalScan	0.025	0.029(+17.1%)	0.029(+18.7%)	0.029(+17.9%)	0.031(+24.8%)	0.031(+26.1%)	0.033(+33.0%)	0.035(+40.7%)	0.037(+50.1%)
SDAC	0.024	0.042(+72.1%)	0.050(+104.4%)	0.054(+121.6%)	0.060(+146.5%)	0.064(+159.6%)	0.066(+169.9%)	0.068(+176%)	0.072(+192.3%)
HomDroid	0.014	0.023(+63.2%)	0.024(+66.0%)	0.023(+59.0%)	0.024(+66.0%)	0.026(+78.5%)	0.028(+94.6%)	0.031(+112.7%)	0.032(+123.8%)
Xmal	0.023	0.025(+11.6%)	0.028(+23.5%)	0.027(+20.4%)	0.027(+18.2%)	0.028(+21.7%)	0.028(+25.3%)	0.029(+28.4%)	0.030(+30.6%)
RAMDA	0.054	0.061(+14.0%)	0.069(+27.6%)	0.075(+38.7%)	0.081(+50.0%)	0.09(+67.1%)	0.093(+73.4%)	0.099(+84.2%)	0.104(+93.9%)
MSDroid	0.072	0.078(+8.4%)	0.080(+11.7%)	0.082(+14.5%)	0.085(+18.4%)	0.093(+29.2%)	0.098(+37.3%)	0.106(+47.4%)	0.111(+55.1%)

A.3 AUT, ASR, and APR

In assessing a classifier's resilience against temporal degradation, we employ the Area Under Time (AUT) metric as suggested by [77]. AUT is formally given by:

$$AUT(f, N) = \frac{1}{N-1} \sum_{k=1}^{N-1} \frac{f(k+1) + f(k)}{2},$$

where f represents the chosen performance metric (such as F1 score or True Positive Rate) and N denotes the count of malware evolution period. AUT values range between $(0, 1)$, where 1 indicates the classifier retains consistent performance across time.

To evaluate a classifier's robustness against adversarial attacks, we use two primary metrics: the attack success rate (ASR) and average perturbation ratio (APR) metrics utilized by [59]. They are mathematically defined as:

$$ASR = \frac{N_{success}}{N_{total}}, \; APR = \frac{F_{modified}}{F_{total}}.$$

Here, $N_{success}$ refers to the number of adversarial examples that successfully deceive the classifier. N_{total} represents the entire set of adversarial samples. Meanwhile, $F_{modified}$ is the number of input features changed by the adversarial intervention, and F_{total} signifies the total number of features in the input. The ASR values fall within

Table 17. The effectiveness of the selected approaches using different-sized training data.

Approach \ Data Size	100%	50%	10%
Drebin	0.722	0.714 (-1.2%)	0.643 (-10.9%)
MamaDroid	0.661	0.623 (-5.7%)	0.452 (-31.5%)
Mclaughlin et al.	0.714	0.627 (-12.2%)	0.140 (-80.4%)
HinDroid	0.731	0.716 (-2.0%)	0.618 (-15.5%)
DeepRefiner	0.657	0.577 (-12.1%)	0.327 (-50.2%)
Kim et al.	0.782	0.742 (-5.1%)	0.611 (-21.8%)
MalScan	0.684	0.653 (-4.6%)	0.526 (-23.1%)
SDAC	0.522	0.496 (-5.0%)	0.474 (-9.1%)
HomDroid	0.734	0.675 (-8.0%)	0.586 (-20.1%)
Xmal	0.698	0.668 (-4.2%)	0.613 (-12.2%)
RAMDA	0.636	0.614 (-3.4%)	0.547 (-14.0%)
MSDroid	0.648	0.563 (-13.0%)	0.424 (-34.6%)

the interval $(0, 1)$; a value of 1 suggests the classifier is entirely susceptible to adversarial attacks. Similarly, APR values lie within $(0, 1)$. A higher APR indicates that evading the classifier becomes increasingly challenging.

A.4 Additional Results

Table 17 presents the detailed results of training data size analysis. In Sec. 5.2, we normalize the results of the selected approaches based on the use of 100% training data size for clarity. For instance, the normalized result for Drebin, using 50% of the training data, is computed as $\frac{0.714}{0.722} = 0.988$. These results highlight the significant impact that the volume of data has on the effectiveness of the selected approaches.

For the analysis of malware evolution, the results of AUT(TPR, N) and AUT(FPR, N) are reported in Table 16 (N is also set as 0, 3, 6, 9, 12, 15, 18, 21, and 24 months). We observe that as malware evolution time increases, there is a consistent decrease in AUT(TPR, N) for the selected approaches. This trend suggests that as malware evolves, more malware samples are misclassified as benign. Simultaneously, an increase in AUT(FPR, N) is noted, indicating that more benign samples are misclassified as malware. Overall, the performance of malware detection approaches degrades with the increase of malware evolution time.