Unraveling the Key of Machine Learning Solutions for Android Malware Detection

Jiahao Liu
National University of Singapore

Jun Zeng
Huawei

Fabio Pierazzi
King's College London

Lorenzo Cavallaro
University College London

Zhenkai Liang
National University of Singapore

ABSTRACT

Android malware detection serves as the front line against malicious apps. With the rapid advancement of machine learning (ML), ML-based Android malware detection has attracted increasing attention due to its capability of automatically capturing malicious patterns from Android APKs. These learning-driven methods have reported promising results in detecting malware. However, the absence of an in-depth analysis of current research progress makes it difficult to gain a holistic picture of the state of the art in this area.

This paper presents a comprehensive investigation to date into ML-based Android malware detection with empirical and quantitative analysis. We first survey the literature, categorizing contributions into a taxonomy based on the Android feature engineering and ML modeling pipeline. Then, we design a general-propose framework for ML-based Android malware detection, re-implement 12 representative approaches from different research communities, and evaluate them from three primary dimensions, *i.e.*, effectiveness, robustness, and efficiency. The evaluation reveals that ML-based approaches still face open challenges and provides insightful findings like more powerful ML models are not the silver bullet for designing better malware detectors. We further summarize our findings and put forth recommendations to guide future research.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

Android malware detection, machine learning, empirical analysis, quantitative analysis

1 INTRODUCTION

Since 2011, Android has become the best-selling operating system (OS) on smartphones [11], captivating billions of users globally. Its worldwide popularity and importance, however, make it a primary target for cybercriminals [5, 12]. These cyber-attacks have resulted in not only privacy breaches but also financial losses. As such, there is a pressing demand to detect Android malware before its installation, causing security concerns.

Android malware detection takes as input an Android Package Kit (APK) and outputs a probability of the APK being malicious. Traditional methods involve manual analysis of suspicious APKs and summarization of malicious patterns as hand-crafted rules [34, 37], but this strategy is not scalable or effective. It falls short in handling the surfeit of new Android software produced annually and proves ineffective against previously unknown malware. Instead, recent advancements leverage machine learning (ML) to improve the scalability and accuracy in Android malware detection [14, 18, 49, 68, 71, 83, 98, 99, 104]. A common paradigm is extracting features from APKs (*e.g.,* permissions and API calls), encoding these features into numeric vectors, and applying ML models (*e.g.,* support vector machine and neural network) to automatically distinguish malware from benign apps, *aka.,* goodware.

Over the past decade, Android malware detection has experienced increased attention from various communities, such as security, software engineering, and machine learning. Much attention has been given to exploring various combinations of APK features and ML models. The trend to date is primarily driven by advancements in ML models (*e.g.*, graph neural network [49]) and analogies drawn from other well-studied fields (*e.g.*, social network [98]). Most existing approaches report high F1 scores (up to 0.99). Such promising results motivate us to ask a number of important research questions. For example: How does the existing literature represent and incorporate diverse features into ML models for Android malware detection? How do different approaches compare when evaluated on the same datasets, metrics, and toolchains? Are current ML-based approaches sufficient to meet the detection requirements in real-world scenarios? Does a more powerful ML model necessarily yield better detection results? Does incorporating more features to describe app behaviors always come with better results? Is there a positive correlation between detection effectiveness and computational efficiency?

In this paper, we aim to investigate the current state of ML-based Android malware detection and offer an in-depth understanding of this field with empirical and quantitative analysis. Given the significant growth of app availability — as seen with Google Play's app count reaching 3.72 million, up 38% from 2022 [6] — this study focuses on assessing approaches that are scalable to large datasets and practical in real-world use cases. To conduct the investigation, we begin with dissecting the ML-based Android malware detection pipeline into three phases: APK characterization, feature representation, and ML modeling. Guided by this taxonomy, we aim to empirically analyze the related literature to explore how current approaches detect Android malware. Afterwards, we experimentally compare the results from a number of representative approaches, allowing us to understand the advancements and challenges encountered in the field. While instructive, several challenges are identified when conducting the comparison.

- *Existing experimental results are not directly comparable.* Previous approaches are usually evaluated on datasets of different sizes, goodware-to-malware ratios, and training-to-testing ratios [18, 51, 63]. 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 to develop detectors, which introduce ambiguity — whether the promising results stem from the novelty of the methodology or not [92].

- *Real-world scenarios, e.g., malware evolution, obfuscation, and adversarial attacks, are not often fully considered in many existing studies.* Android malware detectors face a rapid threat landscape [22], with malware continually evolving to evade detection. The growing usage of obfuscation also makes malware harder to detect as its malicious intentions are hidden [17]. Additionally, ML models can be tricked by intentionally perturbed inputs [25, 45, 91]. Although the impacts of these scenarios have been studied [26, 29, 81], a comprehensive assessment is often missed, leading to a gap in understanding the primary challenges in real-world malware detection.
- *Most methods overlook reporting the end-to-end efficiency from feature engineering to ML modeling.* With the exponential growth of apps in sizes and complexities, feature extraction gradually becomes notably time-consuming — for example, it can take up to 30 minutes to gather API paths from one app [101]. 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 in Android malware detection.

To tackle these issues, we design a general-purpose framework, FrameDroid, to assess 12 representative approaches from three research communities, namely, security [18, 49, 58, 69, 71, 101, 104], software engineering [96, 98, 99], and machine learning [51, 63]. Particularly, we ensure that the same task (*e.g.,* feature extraction) across different methods is performed using the same toolchain to eliminate potential evaluation discrepancies. Moreover, FrameDroid is modular and configurable, facilitating the development of new malware detectors and the design of various evaluation scenarios. For evaluation, we randomly sample 221,310 apps from the public AndroZoo [15] repository, covering the period from 2011 to 2020. The goodware-to-malware ratio is set as 10% to mimic a realistic setting [81]. To make the experiments mirror real-world conditions, we evaluate the selected approaches using standardized metrics, focusing on the following aspects: the effectiveness and efficiency in malware detection, the robustness against app evolution and obfuscation, and the resilience against adversarial attacks.

Through empirical and quantitative analysis, we draw a holistic picture of ML-based Android malware detection. Specifically, APK characterization and ML modeling remain central themes in this field. Many recent detectors achieve comparable results under the same experimental settings. However, their effectiveness in detection still falls short in more challenging scenarios, such as those with limited data volumes or under adversarial attacks. Moreover, we discover that powerful ML models are not a silver bullet for better detection outcomes. Also, while APK features are important to depict apps, sometimes adding more features counterintuitively leads to negative impacts. Another interesting observation is that increased computational overhead is not a reliable indicator of enhanced detection capabilities. Future work should pay more attention to designing robust and practical detectors for real-world usage, with balancing effectiveness and efficiency in mind. (More discussions can be found in Sec. 5).

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 develop a general-purpose framework, FrameDroid, to facilitate the implementation and evaluation of various Android malware detectors. For comparison in realistic settings, we collect and open-source the largest Android app dataset to date, both in size and time span. The framework, along with the dataset, will be released upon acceptance.
- We offer a comprehensive comparative analysis of 12 representative approaches using FrameDroid, focusing on assessing their effectiveness, robustness, and efficiency, from which we make new insights and enhance the understanding of ML-based Android malware detection.

2 MACHINE LEARNING BASED ANDROID MALWARE DETECTION

Android malware detection involves two fundamental steps: characterizing APKs and identifying malicious patterns. Recent trends have seen a shift towards using static feature extraction for APK profiling with reverse engineering [68, 83]. Contrary to approaches that model app behaviors via execution emulation — a process that is time-consuming and resource-intensive [27, 31], static analysis proves to be more efficient and scalable [64]. As such, we exclude approaches that are inherently slow and hard to scale, *e.g.,* dynamic analysis and symbolic execution. For malicious pattern identification, two kinds of approaches have been proposed: (a) rule-based and (b) ML-based. The former requires experts to manually formulate detection rules [34, 37] based on malware characteristics. In contrast, ML models have the capacity to automatically learn patterns from features [18, 63, 71, 98], which lends itself to scalability and adaptability. This has led to the growing popularity of ML models in Android malware detection [83]. In this paper, we focus on ML-based approaches that prioritize static feature extraction for detecting Android malware. Our choice is further validated by recent developments in the field, as discussed in Sec. 6.

Rather than analyzing each approach individually, our methodology is to deconstruct the workflow of ML-based Android malware detection into three distinct phases: APK characterization (Sec. 2.1), feature representation (Sec. 2.2), and machine learning modeling (Sec. 2.3). Following this, we collate and summarize the key techniques employed in each phase by investigating existing approaches, thereby offering a thorough overview of how ML models are applied in Android malware detection.

2.1 APK Characterization

Input from APK. An APK is a compressed archive that contains an app's codebase, resources, and auxiliary files. An APK comprises several essential types of files and folders [36]. *Manifest*: Serving as a descriptor, the manifest provides metadata about the app, detailing elements like the package name, permissions, and hardware components. *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*: Within an APK, native libraries present as shared object files that provide some necessary services for apps (*e.g.,* Webkit), which assists and optimizes

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

Figure 1: APK files and their corresponding features.

the app's execution. *Resource*: This category contains static assets and non-compiled resources integral to the app, such as graphics, layout schematics, and sequences for animations.

APK features. Researchers often deploy static analysis to extract and distill pertinent features. *Manifest* and *Resource* files often adhere to well-defined structures, such as XML format. The features from these files can be efficiently extracted with regular expressions or dedicated XML parsers. In contrast, the *Dex* and *Library* consist of several binary files. Extracting features from them necessitates a deep dive with reverse engineering tools. *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 [7] are helpful in disassembling the native libraries into assembly codes, which facilitates analysis of the native services utilized by apps.

Figure 1 delineates the relationship between APK file/folder and the features derived from them. In the subsequent section, these features are elaborated in detail.

[$\mathbb{M}$] *Hardware Component.* Android apps necessitate the use of certain hardware components (*e.g.,* camera) to execute particular functions (*e.g.,* taking photos). The request for access to specific hardware components carries distinct security implications, as the utilization of certain hardware combinations often indicates potentially harmful behavior [18]. For instance, an app utilizing the camera and network connection may have the capability to monitor user activities and transmit this data to remote servers. To identify such potential malicious behaviors, several approaches [18, 58, 95, 104] check the hardware components requested by an app.

[$\mathbb{M}$] *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 to enter the app. 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 [8]. Inspired by this, application component has been utilized to capture similar fingerprints in Android malware [18, 58, 95, 104].

[$\mathbb{M}|\mathbb{D}$] *Intent.* As the primary ways of communication among components, intents connect various Application Components and delineate the standard operations the app can perform. They are pivotal in initiating Activities, managing the lifecycle of Services, and delivering broadcast information to Broadcast Receivers. Malware frequently monitors these intents to trigger malicious actions, such as activating pre-configured malicious activity [113]. Approaches [39, 63, 103] attempt to capture these potential malicious behaviors by analyzing corresponding intents.

[$\mathbb{M}$] *Permission.* Android employs a permission-based mechanism to regulate application access to sensitive data and restricted actions, such as accessing contact information or establishing a connection with a paired device. For an app to carry out particular actions, it must obtain the requisite permissions at the point of installation or during runtime [20, 21]. The set of permissions required by an app can thus offer insights into its intended behaviors. Particularly, malware often demands permissions that are typically unnecessary for benign apps, enabling them to execute malicious actions [34]. Consequently, numerous malware detectors [18, 49, 58, 96, 104] capture the differences to distinguish malware.

[$\mathbb{D}$] *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 [78]. For instance, the invocation of *sendTextMessage()* suggests that the app is likely to send a text message. Furthermore, API calls are often connected to form a program graph, where each node signifies a method, and each edge denotes a method invocation [61], illustrating the app's structural information [98]. 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 [18, 58, 96] detect sensitive API calls (*e.g., getDeviceId()*) to estimate the probability of an app being malicious. In contrast, other solutions [49, 51, 70, 98, 101, 107] venture deeper, examining the relationships between API calls to capture an app's intended behaviors.

[$\mathbb{D}|\mathbb{L}$] *ByteCode and Opcode.* Consistent with previous research [32, 65, 88, 104], we treat both the *raw bytecode* and the *assembly code* derived from the *Dex* and *Library* as ByteCode representations. Specifically, ByteCode is structured in 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` Opcode signifies 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 comprehensive view of the apps' intended behaviors [58]. Recent research attempts to represent Bytecode and Opcode in various formats, such as image [16, 52, 71, 100] and text [56, 104, 105], to capture apps' semantics.

[$\mathbb{D}$] *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 [18]. For instance, malware often sets up network socket connections to communicate with remote servers, resulting in the presence of the string `socket` within the codebase. There have been a number of works [18, 58, 115] using code strings to identify potential illegal operations.

[$\mathbb{R}$] *Resource Information.* Apps utilize resources to house traditional files and static elements, such as bitmaps, layout definitions, 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.

2.2 Feature Representation

APK characterizations offer multi-perspective views of an app's behavior. Before feeding these features into ML models, they must be encoded into a format that is readily interpretable by these models. According to APK feature representations, the encoding process can be organized into four main categories: categorical, image-based, text-based, and graph-based. Figure 2 illustrates the relationships between these encoding strategies and specific APK features. In the subsequent section, we discuss these encoding strategies in detail.

Categorical encoding. In an APK, 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 [18, 38, 51, 62, 63, 96, 108]. On the other hand, another line of research [58] calculates the frequency of each feature to obtain a vector of numerical values.

Image-based encoding. Representing specific features as images and subsequently leveraging image processing techniques is a well-established approach in Android malware detection. Specifically, bytecode and opcode sequences are often visualized as images to describe apps' behaviors [32, 33, 52, 71, 100]. Notably, DexRay [32] 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. [109] combine API calls with protection levels as an RGB image.

Text-based encoding. Similar to image representation, text-based encoding is also a widely used strategy in Android malware detection. Many existing methods [56, 57, 85, 101, 104] 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 [56, 57, 101] utilize word embedding techniques, such as Word2Vec [73], to extract semantic information included in the API calls.

Graph-based encoding. Recently, graph structure has been widely adopted to represent apps' semantics [49, 61]. One notable research direction is to utilize program graphs as the basis to model APK behaviors [49, 79, 80, 98, 99]. 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 [51, 53]. When represented as a graph, various techniques, such as DeepWalk [82] and Graph2Vec [75], are employed to extract apps's structural information for malware detection.

2.3 Machine Learning Modeling

After encoding these features as numerical vectors, machine learning models can be leveraged to identify malicious patterns from them. Following previous studies [83, 112], 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 [111]. 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 in Android malware detection.

Categorical	⇒	*Hardware Component* | *API Call* | *ByteCode* | *Resource Information* | *Opcode* | *Code String* | *Application Component* | *Permission* | *Intent*
Image-based	⇒	*ByteCode* | *Opcode* | *API Call*
Text-based	⇒	*ByteCode* | *Opcode* | *API Call*
Graph-based	⇒	*Program Graph* | *API Call*

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

TML models. Given the features extracted based on expert knowledge, TML models are commonly employed to discern patterns from these features. The Support Vector Machine (SVM) can find one hyperplane that effectively separates the high-dimension data points with varying labels. This capability has made it a popular choice to detect Android malware [18, 41, 51, 87, 101]. K-Nearest Neighbors (KNN) has also been applied in Android malware detection as seen in studies [14, 97–99]. 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 [69, 116].

DL models. Recently, DL models have exhibited strong capability in modeling malware behaviors. 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 [28, 58, 63, 70, 85, 114]. The Recurrent Neural Network (RNN) [72] 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, several solutions [93, 104, 105] leverage RNN to explore the temporal dependencies embedded in these features. The Convolutional Neural Network (CNN) [46] is equipped with multiple convolutional and pooling layers, enabling it to recognize contextual information derived from low-level features [42]. Reflecting its efficacy, CNN has been extensively employed to extract malicious patterns from image-based features in Android malware detection [38, 47, 52, 53, 57, 71]. Given the graph representation of APK features (*e.g.,* program graphs), Graph Neural Network (GNN) [59] can facilitate malware detection [35, 40, 49, 69]. 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 [63, 106, 117].

In addition, existing research [16, 84, 94] also tries to explore the potential of other DL models, like generative adversarial network (GAN) [43] and deep belief network (DBN) [50]. For instance, Amin

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 [18]	●	●	●	●	●	○	○	●	○	○	Categorical	SVM
MamaDroid [69]	○	○	○	○	●	○	○	○	●	○	Graph-based	RF
Mclaughlin et al. [71]	○	○	○	○	○	○	●	○	○	○	Image-based	CNN
HinDroid [51]	○	○	○	○	●	○	○	○	●	○	Graph-based	SVM
DeepRefiner [104]	●	●	●	●	○	●	○	○	○	●	Text-based	LSTM
Kim et al. [58]	●	●	●	●	●	○	●	●	○	○	Categorical	MLP
MalScan [98]	○	○	○	○	●	○	○	○	●	○	Graph-based	KNN
SDAC [101]	○	○	○	○	●	○	○	○	●	○	Categorical	SVM
HomDroid [99]	○	○	○	○	●	○	○	○	●	○	Categorical	KNN
Xmal [96]	○	○	○	●	●	○	○	○	○	○	Categorical	MLP
RAMDA [63]	○	○	●	●	●	○	○	○	○	○	Categorical	AE
MSDroid [49]	○	○	○	●	●	○	●	●	●	○	Graph-based	GNN

While multiple ML models may be utilized in individual approaches [69, 98, 99], we only report the model that yields the best effectiveness (*e.g.*, F1-score).

et al. [16] 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.

3 REPRESENTATIVE APPROACH ANALYSIS

To understand the state-of-the-art ML-based Android malware detection, a quantitative analysis of existing literature is indispensable. With the rapid evolution of this field, hundreds of techniques have been proposed and achieved remarkable performance in the past decade. A systematic investigation of relevant works dating back to 2011 is discussed in Sec. 6 to highlight extensive efforts that have been devoted to Android malware detection.

While an ideal scenario is evaluating as many approaches as possible to gain a better understanding of the current research state, 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 strategically focuses on methods that represent the broad spectrum and depth of advancements in the field. In the subsequent section, we first outline the principles guiding our selection and provide a summary of the selected approaches. Then, a comparative analysis of these methods is presented to offer insights into their experimental designs.

3.1 Selection Criteria

Cover various communities. Android malware detection stands as an interdisciplinary domain, drawing contributions from diverse communities, including security, software engineering, and machine learning. Unfortunately, a notable separation is often observed among these communities — the solutions in one community often only compare with others from the same community, which hinders potential collaborative advancements. Thus, we want to incorporate approaches from a broad range of communities to bridge the gap and foster greater collaboration.

Explore an extensive spectrum of techniques within the detection pipeline. As identified in Sec. 2, a wide range of technique combinations exists across different phases of the detection pipeline. This study aims to explore 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 Android malware detection.

Reflect research progress. Considering the evolving landscape of this field, where novel methodologies continually emerge, we prioritize approaches that introduce new techniques or achieve remarkable performance, such as proposing a new feature representation or employing a new learning architecture.

Emphasize representative approaches over specific papers. Many approaches share similar techniques, commonly 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. Therefore, we focus on distinct and representative strategies that offer more significant contributions to this field.

3.2 Selected Approaches

Adhering to the selection criteria, we identify 12 representative approaches from hundreds of available solutions to analyze the current state of ML-based Android malware detection, as presented in Table 10. These methods are carefully chosen to represent different communities — encompassing 7 from security, 3 from software engineering, and 2 from machine learning. This selection guarantees a broad range of techniques employed in the detection process, including 10 APK features, 4 feature representations, and 8 ML models. Importantly, the selected approaches stand out either by demonstrating promising performance or bringing novel techniques to the discipline, *e.g.*, GNN [49] and CNN [71]. Additionally, we have made a concerted effort to ensure that the selected solutions are not variants or combinations of existing ones. For instance, while several methods [35, 51, 107] utilize heterogeneous information graphs to model APKs, we spotlight the pioneering approach [51] that first introduces this concept in this area. In the remaining part of this section, we introduce these selected approaches, integrating the information presented in Table 1.

Drebin. As a lightweight Android malware detector, Drebin [18] first collects APK features, such as permissions, intents, and API

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 [18]	129,013	2010-2012	2:0:1	4%	●	✘	✘	✘	✔	Androguard
MamaDroid [69]	43,940	2010-2016	9:0:1	50%	●	✔	✘	✘	✔	Soot
Mclaughlin et al. [71]	27,395	—	—	50%	◐	✘	✘	✘	✔	BackSmali
HinDroid [51]	2,334	2017-2017	4:0:1	60%	◐	✘	✘	✘	✘	APKTool
DeepRefiner [104]	110,440	—	**8:1:1**	57%	●	✘	✔	✔	✘	APKTool
Kim et al. [58]	41,260	—	**3:1:1**	50%	◐	✘	✔	✘	✘	APKTool
MalScan [98]	30,715	2011-2018	9:0:1	50%	●	✔	✘	✔	✔	Androguard
SDAC [101]	70,142	2011-2016	8:0:2	50%	●	✔	✔	✘	✘	FlowDroid
HomDroid [99]	8,198	—	9:0:1	40%	●	✘	✘	✘	✘	Androguard
Xmal [96]	35,690	—	7:0:3	43%	○	✘	✘	✘	✔	Androguard
RAMDA [63]	58,483	—	19:0:1	50%	○	✘	✘	✔	✔	APKTool
MSDroid [49]	81,790	2010-2015	4:0:1	37%	○	✔	✔	✘	✔	Androguard

calls, using static analysis. These features are then converted into a binary vector to signify their existence or absence. From the data, an SVM model is trained to detect malware.

MamaDroid. MamaDroid [69] pioneers the use of Markov chains to depict apps' behaviors. With the Markov chain constructed from the sequence of API calls, the transition probabilities between these calls are computed to serve as features. These features then inform an RF model to detect Android malware.

Mclaughlin et al. This technique [71] presents the leading edge in image-based Android malware detection approaches. By transforming opcode sequences into images via one-hot encoding, it leverages a CNN model to distinguish malware from benign apps.

HinDroid. Hou et al. [51] 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 [89]. A multi-kernel SVM is further applied to recognize malicious patterns.

DeepRefiner. DeepRefiner [104] 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 a long short-term memory (LSTM) model to capture the method-level and application-level semantics.

Kim et al. Kim et al. [58] innovatively 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 Android malware identification.

MalScan. This study [98] treats app program graphs as social networks, where API calls are treated as nodes, and the relationships between them are presented as edges. The system evaluates the centrality of sensitive API calls to derive features and then feeds them into a KNN model to detect malware.

SDAC. The algorithm [101] attempts to cluster API calls based on their contextual information extracted by Word2Vec [73] from API call sequences. These resulting clusters act as features to represent APKs. An SVM model is then used to capture malicious patterns.

HomDroid. The method in [99] zeroes in the suspicious components of malware by calculating the homophily within its program graph, paving a novel path for malware detection. From the malicious subgraph, 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 [96] 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 [63] is the state-of-the-art approach that employs Autoencoder to derive a resilient representation of APKs with features such as API calls and intents. Then, the representation is fed into an MLP to detect Android malware.

MSDroid. MSDroid [49] is the cutting-edge in utilizing GNN to detect Android malware. Initially, it breaks down the program graph into subgraphs rooted with sensitive API calls. Then, it leverages a GNN to capture essential information for malware detection.

3.3 Comparative Study

Comprehensively and comparatively analyzing these representative approaches from various angles provides an insightful lens for understanding the current advancements in ML-based Android malware detection. In this section, we conduct a comparative review of the selected approaches, focusing on 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.

Effectiveness. Effectiveness is the most important criterion for any detection technique, and all the selected approaches evaluate this aspect. However, the experimental settings — dataset size, time span, dataset partition, and malware ratio — vary dramatically across these approaches, as shown in Table 2, making direct comparisons challenging. 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 [51] to 129,013 [18], 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 [49, 63, 96]. This oversight is alarming, raising concerns about potential over-fitting and over-optimistic performance indicators. In the wild, malware generally accounts for around 10% of cases [81]. However, this ratio in testing datasets significantly varies across studies (*e.g.,* from 4% [18] to 60% [51]), which makes it difficult to reveal the true effectiveness of these approaches.

Robustness. Android malware detection faces three major challenges: malware evolution, obfuscation, and adversarial attacks [30, 81]. Specifically, Android malware detectors routinely operate in hostile and dynamic contexts [22], where malware constantly evolves to evade detection. Also, obfuscation has been widely adopted by attackers to conceal their malicious operations [17]. Additionally, the inherent susceptibility of ML models to adversarial attacks [19, 61] complicates the detection process. In our estimation of the robustness of the selected approaches, as detailed in Table 2, we observe that these selected approaches frequently miss one or more of the real-world challenges. This omission complicates the assessment of their true effectiveness in real-world deployments.

Efficiency. To measure a new technique, 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.

Other considerations. Table 2 additionally provides information on whether the selected approaches make their artifacts available and the specific toolchains they utilize. We note that nearly half of these approaches do not release their artifacts, posing significant challenges to reproducibility. Furthermore, the toolchains used by these approaches are diverse, such as Androguard [1], APKTool [3], and BackSmali [4]. Such heterogeneity in toolchain selection complicates the direct comparison of these methods' effectiveness, as different toolchains can substantially impact the outcomes. Combining with the aforementioned analysis, there is a pressing need for a general-purpose framework that not only unifies the development process but also supports multi-aspect evaluation scenarios for ML-based Android malware detectors.

4 QUANTITATIVE ANALYSIS

One of our primary contributions is conducting a comprehensive and quantitative analysis of the selected representative approaches.

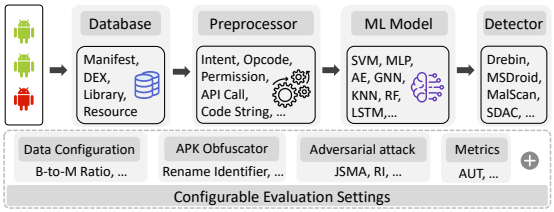

Figure 3: The architecture of FrameDroid.

This analysis aims to estimate and unravel the effects of various real-world scenarios, such as different data sizes, goodware-to-malware ratios, and the presence of adversarial attacks, on the performance of these methods. Such experiments offer valuable insights into the current state of ML-based Android malware detection. Specifically, within this section, we re-implement these 12 approaches utilizing our general-purpose framework, FrameDroid (Sec. 4.1). We then assess their effectiveness (Sec. 4.3), robustness (Sec. 4.4), and efficiency (Sec. 4.5) on our crafted dataset — the largest to date in terms of both time and size (Sec. 4.2).

4.1 FrameDroid

Figure 3 illustrates the architecture of FrameDroid. The procedure initiates with a collection of Android apps, from which a set of widely recognized features is extracted and stored in a feature database. These features are then processed by a preprocessor, being transformed into a numerical format suitable for ML models. After that, an ML model is trained and evaluated for detecting malware. FrameDroid is also equipped with adaptable experimental settings to support a wide range of real-world scenarios.

FrameDroid consists of three key modules: *feature database*, *preprocessor*, and *ML model*. *Feature database* organizes the extracted features categorically, including Manifest, ProgramGraph, DisassembledCode, and SharedLibrary (details are in A.1), which can be easily extended to incorporate new features. The role of *preprocessor* is to retrieve and encode features before feeding them into *ML model*. This module can be tailored to support different detectors, granting flexibility in feature selection and usage. *ML model* integrates widely used ML models, *e.g.,* RF, SVM, KNN, MLP, LSTM, CNN, GNN, and AE. It also supports model customization — for example, one can easily add new neural networks of different architectures. To develop a new detector, one only needs to customize *preprocessor* to select corresponding features from *feature database* and feed them into *ML model*.

FrameDroid provides configurable experimental settings to support different real-world scenarios. Parameters, such as goodware-to-malware ratios and training data sizes, are all adjustable. Different configurations enable the creation of diverse evaluation scenarios. FrameDroid also includes an APK obfuscator [17], which is instrumental in producing different types of obfuscated samples, *e.g.,* code modification and identifier renaming. Moreover, this framework incorporates an adversarial samples generation mechanism from AndroidHIV [29], allowing the assessment of models' resilience to adversarial attacks.

Utilizing FrameDroid, we have re-implemented and conducted experimental analysis on the 12 representative approaches, as elaborated in Sec. 3. Detailed information regarding the implementation of these approaches can be found in A.3.

Table 3: Evaluation Dataset Statistics. This dataset consists of 221,310 Android applications, i.e., 22,870 malicious and 198,440 benign apps, spanning ten years from 2011 to 2020. The unit used for measuring APK size is megabytes (MB).

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

Table 4: Data distribution of four sub-datasets. 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

4.2 Dataset

For an evaluation that mirrors real-world scenarios and supports a multi-dimensional assessment, it is crucial that the dataset adheres to the following criteria [81]. *Market Diversity*: Since Android apps are distributed across various app stores, it is better to sample apps from multiple platforms to ensure comprehensive representation. *Grayware*: Owing to the uncertain nature of grayware, incorporating it might skew the results. To prevent potential biases which could misrepresent a method's performance, grayware should be excluded. *Time Distribution*: To study malware evolution [24, 60] in Android malware detection, the dataset should contain samples over a long time range. *Malware Ratio*: To mimic actual conditions, the dataset should have a malware ratio that aligns with the estimated 10% observed in the wild [81].

Several public repositories, such as AndroZoo [15], Drebin [18], and AMD [66], provide access to collections of Android apps. Among these, Androzoo stands out as a continually expanding repository, aggregating apps from various platforms, like Google Play, PlayDrone, VirusShare, and AppChina. In contrast, other repositories typically consist of apps from fixed periods and are sourced from a relatively smaller number of platforms. Consequently, we choose to collect our dataset from AndroZoo, which ensures our evaluation is more comprehensive and representative. During the selection process, we use the positive anti-virus alerts from VirusTotal [13], denoted as p, to filter out grayware. Following previous studies [74, 81], apps with $p \geq 4$ are categorized as malicious, while those with $p = 0$ are considered benign. The dataset covers apps released between 2011 and 2020. We further ensure that the malware ratio for each month approximates the 10% target, aligning with *Malware Ratio*. We have successfully obtained a dataset of 221,310 Android apps, i.e., 22,870 malicious and 198,440 benign apps, which is summarized in Table 3.

Settings. To support certain evaluation scenarios, we also construct several sub-datasets from the aforementioned main dataset. For each sub-dataset, apps are randomly sampled to form training, validation, and testing sets. Table 4 shows the data distribution of these four sub-datasets. Importantly, we prioritize the apps in training, validation, and testing sets from different years to ensure comprehensive representation. For instance, for ❶, each year contributes 1440 benign and 160 malicious apps for training, 180 benign and 20 malicious apps for validation, and the same for testing. Duplicate apps are also avoided across these sets.

Table 5: The effectiveness of these 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 denotes the training set, and *Ts* represents the testing set. ❶ - ❹ correspond to the four scenarios in Table 4. The top 3 results for each scenario are highlighted in bold.

4.3 Effectiveness

We evaluate the selected methods under various experimental conditions. Specifically, we investigate how various factors — goodware-to-malware ratio in training and testing sets, training set size, and APK feature — affect the effectiveness of these approaches.

Goodware-to-malware ratio. The effectiveness of ML-based malware detectors is heavily influenced by the distribution of malware in the training and testing sets. We aim to investigate how this ratio affects the selected approaches. Particularly, we consider two ratios, *i.e.*, 10%, and 50%, in both training and testing sets. These choices are inspired by the estimated 10% observed in the wild [81] and the 50% widely adopted in previous studies [63, 69, 98]. The ❶ - ❹ in Table 4 present these four scenarios in our consideration.

Table 5 summarizes the results in terms of F1-score and Accuracy. The results are arranged sequentially from left to right, corresponding to scenarios ❶ through ❹. We observe that all the approaches produce promising results under scenario ❹, characterized by a `1:1` goodware-to-malware ratio in both training and testing sets. This demonstrates that these approaches can effectively detect malware under ideal conditions, where goodware and malware are balanced. However, when the malware ratio in the testing set is adjusted to 10% (❶ and ❸), there is a marked decline in the F1-score across all methods. It is evident that most of these approaches suffer from

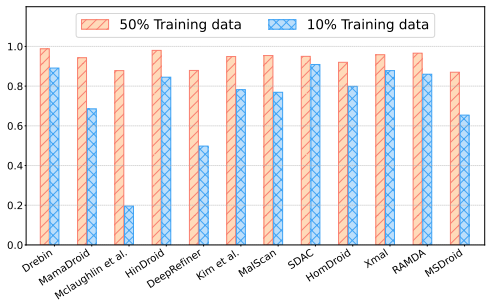

Figure 4: Effectiveness of the selected approaches using different sizes of training data.

performance degradation when faced with a realistic goodware-to-malware ratio. In addition, upon analyzing the top three results, we find that DL-based approaches appear to outperform traditional ML-based methods when the 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 marked advantage over traditional ML-based methods.

Training set size. The training set's size is a critical determinant of the effectiveness of ML classifiers [86]. However, securing a large, high-quality training set is often infeasible due to the significant costs of data collection and labeling. To study the impact of training set size on malware detection, we down-sample the training set in Table 4(❶) to 50% and 10% of its initial volume, while ensuring that the malware ratio remains consistent at 10%.

Figure 4 shows how these detectors' performance changes when the training set size is reduced. To offer a clearer view, we normalize the F1-scores of these methods based on their performance obtained with the entire original dataset. The detailed results and the normalization process are provided in A.6. The figure clearly shows that reducing the training set size leads to performance degradation across all selected approaches, underscoring the principle that more data enhances capturing malicious patterns. Interestingly, Mclaughlin et al. [71] and DeepRefiner [104] display a greater sensitivity to training set size compared to others. One possible explanation is that, unlike other methods that heuristically select features from apps, these two approaches take original apps' bytecode as input, requiring more data to learn patterns from raw data as opposed to hand-crafted features. This finding highlights the importance of feature selection in ML-based Android malware detection, especially in scenarios where data availability is limited.

APK feature. ML-based Android malware detectors often leverage diverse features to enhance detection performance [18, 58, 63]. The underlying reason is that each feature uniquely contributes to characterizing APKs. One question naturally arises: Does the incorporation of more features necessarily enhance a detector's performance? To investigate this, we systematically remove individual features from the original feature set to assess the resulting model performance. For this experiment, it is essential that the feature sets of the chosen approaches are decomposable, allowing the sequential removal of individual features. Accordingly, we spotlight Drebin [18], Kim et al. [58], Xmal [96], and RAMDA [63] as exemplary approaches, owing to their decomposable feature sets. We exclude certain methods whose features are highly intertwined from this study. For instance, MamaDroid [69] and MalScan [98] 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. It is worth noting that the selected methods are representative to conduct this experiment because they cover most features and various models.

Table 6: 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 indicates that the feature are not used in the original work.

Table 6 presents the outcomes of this experiment. From the table, we observe that Drebin [18], Xmal [96], and RAMDA [63] has a performance degradation when some features are removed. Interestingly, Kim et al. [58] deviates from this trend. In fact, even when some features are omitted, its performance exceeds the original results. These two observations underscore that each feature has a distinct contribution to the overall effectiveness of a malware detector. While in some cases, merely expanding the feature set does not guarantee an enhanced performance. This phenomenon underscores the importance of evaluating and justifying the inclusion of each feature in a malware detector.

4.4 Robustness against real-world scenarios

As discussed in Sec. 3, the real-world scenarios — malware evolution, obfuscation, and adversarial attacks — are the main challenges for ML-based malware detectors. While previous works [48, 61, 81] have started investigating detectors' robustness, they often focus on one or two parts of these scenarios. Thus, a thorough assessment of ML-based malware detectors' robustness against a wider range of scenarios is still lacking. To bridge the gap, this section comprehensively re-examines and evaluates the robustness of the selected approaches, offering insights into the current state of ML-based malware detection.

Evolution. Android malware is continuously evolving, and the effectiveness of malware detectors can be affected by this evolution [55, 110]. To quantify the impact of malware evolution on malware detectors,, we utilize the $AUT(f, N)$ metric as introduced by [81], where f denotes the F1-score of a given approach, and N represents the evolution period. We set N to 3, 6, 9, 12, 15, 18, 21, and 24 months in this experiment (see A.4 for details). This metric ranges in (0, 1), where higher values indicate greater resilience of an approach to malware evolution. In the study, we adopt a rolling algorithm over the data from 2011 to 2020 to calculate the $AUT(f, N)$ (see A.5 for the algorithm). 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. These scores are further used to calculate the $AUT(F1, N)$. By averaging the

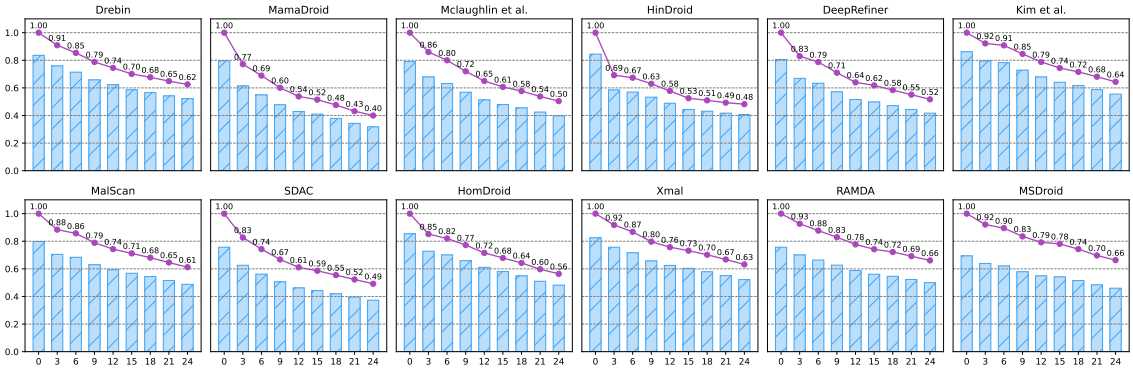

Figure 5: The performance of the selected techniques against diverse malware evolution periods. Columns display the absolute values of $AUT(F1, N)$. The line charts depict the relative percentage of $AUT(F1, N)$ against $AUT(F1, 0)$.

Table 7: The F-score of the selected approaches under different obfuscation strategies [17].

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

$AUT(F1, N)$ values sourced from distinct yearly datasets for each method, we chart the outcomes in Figure 5.

It is clear that malware evolution affects the effectiveness of these selected techniques. Notably, most DL approaches display superior resilience against malware evolution compared to their traditional ML counterparts. Specifically, the majority of DL techniques retain about 60% effectiveness even after two years. In contrast, many traditional ML methods see their F1-scores reduced to 50% within the same time span. This disparity can likely be traced back to the inherent capability of DL methods to discern intricate patterns, which traditional ML might miss. Interestingly, certain DL-based methods, *e.g.*, Mclaughlin et al. [71] and DeepRefiner [104], do not fare as well over time. This could be attributed to their reliance on apps' original bytecode as input, which may complicate the malware pattern learning process.

Obfuscation. As a common practice in Android development, obfuscation is often utilized to safeguard developers' intellectual property and deter reverse engineering attempts [41]. However, as obfuscation typically involves altering an app's code, it can also affect the effectiveness of malware detectors [49]. This part explores the influences of popular obfuscation techniques, *i.e.,* renaming identifiers, encrypting resources, modifying code, and invoking reflection [17], on the selected approaches.

In this study, we apply each of the obfuscation strategies discussed earlier to the testing set outlined 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.

Table 8: 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

The effectiveness of the selected approaches, previously trained on the original training set, is then evaluated on this obfuscated testing set. Table 7 summarizes corresponding results. Clearly, most methods exhibit a decrease in effectiveness when subjected to obfuscation. More specifically, the impact of obfuscation on detectors' effectiveness significantly depends on how the detectors utilize APK features. For instance, Xmal [96] and RAMDA [63] demonstrate greater robustness against obfuscation. This resilience is mainly due to the fact that the employed obfuscation techniques do not change their used features. In contrast, detectors like MamaDroid [69] and MSDroid [49], relying on the code structure of the APK, tend to be more susceptible to the code modification strategy.

Adversarial attack. While ML-based detectors have shown promising results, their susceptibility to adversarial samples remains a concern [44, 54]. We now utilize the dataset from Table 4(❶) to explore the impact of adversarial attacks on the performance of the selected approaches. Mclaughlin et al. [71] and DeepRefiner [104] are excluded since they truncate features at a certain size, making them easily bypassable. Thus, attackers could embed malicious code in the ignored segments to evade detection.

To better investigate the impact of adversarial attacks, we employ two basic strategies: Jacobian Saliency Map Attack (JSMA) and Randomized Input (RI). Utilizing the JSMA approach, we first train a substitute model with the training set, followed by applying JSMA on the substitute model to generate adversarial samples in the testing set. For the RI technique, we randomly change a certain percentage of features in the testing set to produce adversarial examples. Importantly, when crafting adversarial samples, we follow [29, 61] to ensure the feature modifications are domain-mappable and can be repackaged to APKs. Using Drebin as an example, we restrict our modifications to changing feature vectors from 0 to 1, keeping apps' core functionalities unaffected. To

Table 9: 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

measure the robustness of these approaches against adversarial attacks, we use two metrics from [61]: Adversarial Success Rate (ASR) and Adversarial Perturbation Ratio (APR), with their definitions provided in A.4. Both metrics range (0, 1). A higher ASR indicates increased vulnerability to adversarial attacks, while a higher APR suggests greater robustness against such attacks.

By analyzing Table 8, we note that JSMA yields an average ASR of 98.8% across the evaluated approaches, indicating their vulnerability to such basic adversarial attacks. The employed strategies are less effective on MamaDroid [69], HomDroid [99], and RAMDA [63] compared to others. The robustness in MamaDroid and HomDroid could be linked to their unique 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 API calls and program graphs. Additionally, RAMDA's resilience can be attributed to its customized autoencoder, which learns compressed representations of benign apps, making it more defensive to adversarial perturbations. These findings suggest that abstracting feature representations or augmenting models' defensive capabilities could improve malware detectors' robustness to adversarial attacks.

4.5 Efficiency

Efficiency often serves as a primary indicator to measure scalability and applicability. In this section, we scrutinize the performance of the selected methods across datasets from different years, focusing on both feature transformation (*i.e.*, retrieving and encoding) and ML modeling. All experiments are performed on a server equipped with 32-core CPUs operating at 2.10 GHz, 251 GB of physical memory, and two GPUs, each with 32 GB of memory.

Feature transformation. With the pre-extracted features in our database, we assess the time efficiency of the selected methods in retrieving and encoding their required features. Note that we utilize 16 processes to transform the features concurrently.

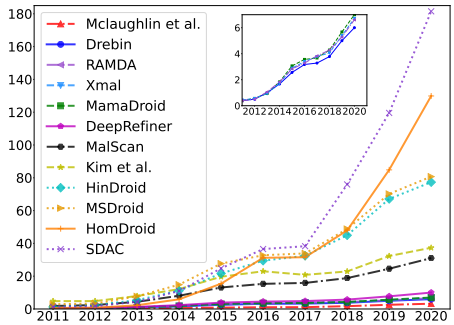

Figure 6: The efficiency of feature transformation of the selected approaches. The x-axis shows the dataset year, and the y-axis indicates the utilized time in hours.

Figure 6 depicts the time taken by these approaches for feature transformation. As time advances, a noticeable increase in the time required for processing is evident, which aligns with the exponential growth in the size and complexity of APKs over the years. Notably, there is a significant variance in the time consumption among different approaches. This is within our expectation since various methods employ distinct features and encoding techniques. For instance, SDAC [101] consumes the most time due to its requirement to query the program graph recursively for API call sequence generation. While methods like Xmal [96] and RAMDA [63] are more time-efficient, as they perform a one-time traversal on the program graph to capture the utilized API calls.

ML modeling. For each yearly dataset, we divide it into training, validation, and testing sets with a ratio of `8:1:1`. We then calculate the time required by the selected methods for model training and testing. Note that we take the early stopping strategy during the training phase for DL models.

Table 9 details the time taken by these approaches. From a horizontal perspective, we observe that DL techniques consistently require more time than their traditional ML counterparts. Furthermore, model complexity directly correlates with its training duration. Analyzing longitudinally, there is a clear trend: as years progress, the time requirements for these methods increase. This

can be attributed to apps growing in complexity and an expanding feature set. When combined with the results of effectiveness (Sec. 4.3), we note that detection effectiveness does not always match resource utilization. Striking a balance between effectiveness and resource efficiency is critical when developing new detectors.

5 FINDINGS

With the empirical investigation and quantitative analysis, 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.

Current ML-based Android malware detectors still face open challenges. While ML models have been evidently effective in detecting malware [49, 58, 63, 69, 98], their effectiveness is still far from satisfactory when faced with challenging scenarios such as limited data volume, rapid malware evolution, and adversarial attacks. Given this context, the field presents substantial opportunities for improvement, particularly in designing robust and practical detectors for real-world usage.

Feature engineering is a critical step towards improving detection performance. Diverse APK features, such as permissions and intents, have been leveraged to identify malware [18, 58, 104]. Specifically, these features profile apps and play a pivotal role in shaping the effectiveness of ML-based detectors. Our analysis reveals that feature engineering helps models to discern malicious patterns when data is limited. However, including more features randomly does not guarantee enhanced performance and is sometimes counterproductive. Thus, researchers should carefully select features to improve the effectiveness and efficiency of detectors.

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 [18, 49, 63, 104]. Our experimental analysis reveals that DL-based approaches appear to be more resilient than traditional ML-based approaches in adapting to malware evolution. However, it also has been observed that DL-based approaches are less effective than traditional ML-based methods when the malware ratio is low. This suggests that utilizing more complex models is not a universal solution for malware detection. When introducing a new model to detect malware, it is crucial to justify its ability to unveil malicious patterns, and the rationale behind its effectiveness.

Both feature abstraction and models' defensive mechanism contribute to detectors' robustness. Adversarial attacks pose a substantial challenge to the robustness of Android malware detection [61, 63]. Our analysis indicates that abstracting features can help models capture more robust malicious patterns [23]. For instance, MamaDroid [69] abstracts program graphs with family and package names, enhancing its robustness to JSMA and RI attacks. Moreover, bolstering the defensive capability of ML models can further amplify detectors' reliability and stability [77].

Detection effectiveness does not positively correlate with efficiency. Through a combined analysis of effectiveness and efficiency, we observe that in Android malware detection, effectiveness and efficiency do not always positively correlate. A detection tool that demands more resources does not necessarily deliver enhanced results. For example, Drebin [18] can achieve competitive detection performance with a relatively low resource consumption.

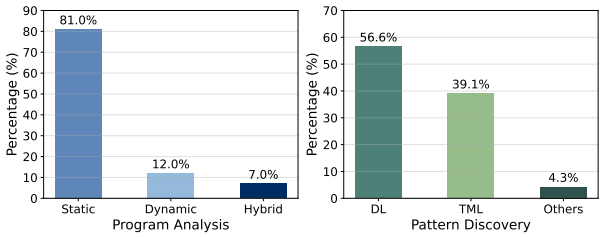

Figure 7: The overall distribution of investigated approaches from 2011 to 2023.

6 DISCUSSION

Systematic investigation. To better understand the efforts dedicated to Android malware detection, we conduct a systematic literature review spanning from 2011 to 2023. Aiming for the inclusion of a broad range of papers, we follow the search strategies used in [67, 68]. Specifically, we perform searches in key digital libraries, such as the ACM 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 7 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.

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 it, we do not include the methods designed to solve a particular challenge like malware evolution. Specifically, there have been several attempts [30, 76, 102, 110] starting to mitigate the challenges we have identified. For instance, the recent APIGraph [110] 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. While promising, 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 outcomes and the reported ones due to different settings, such as

datasets, metrics, and toolchains. To mitigate experimental biases and provide a fair comparison across diverse approaches, 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, adversarial attacks, 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.

7 CONCLUSION

This paper performs the first systematic study of the ML-based Android malware detection literature with empirical and quantitative analysis. We identify challenges 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. Committed to the research community's growth, all the artifacts (code, data, and logs) will be released upon acceptance.

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.]. Harly: another Trojan subscriber on Google Play. https://www.kaspersky.com/blog/harly-trojan-subscriber/45573.
[6] [n. d.]. How Many Apps In Google Play Store? https://www.bankmycell.com/blog/number-of-google-play-store-apps.
[7] [n. d.]. IDA Pro. https://hex-rays.com/ida-pro/.
[8] [n. d.]. Kharon project. https://cidre.gitlabpages.inria.fr/malware/malware-website/dataset/malware_DroidKungFu1.html.
[9] [n. d.]. LibRadar. https://github.com/pkumza/LibRadar.
[10] [n. d.]. PyTorch. https://pytorch.org/.
[11] [n. d.]. Share of Android OS of global smartphone shipments. https://www.statista.com/statistics/236027/global-smartphone-os-market-share-of-android.
[12] [n. d.]. The mobile malware threat landscape in 2022. https://securelist.com/mobile-threat-report-2022/108844.
[13] [n. d.]. VirusTotal. https://www.virustotal.com.
[14] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. Droidapiminer: Mining api-level features for robust malware detection in android. In *International ICST Conference, SecureComm*.
[15] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. Androzoo: Collecting millions of android apps for the research community. In *MSR*.
[16] 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).
[17] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. 2020. Obfuscapk: An open-source black-box obfuscation tool for Android apps. *SoftwareX* (2020).
[18] 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*.
[19] Anish Athalye, Nicholas Carlini, and David Wagner. 2018. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *ICML*.
[20] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: analyzing the android permission specification. In *CCS*.
[21] 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*.
[22] Federico Barbero, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. 2022. Transcending transcend: Revisiting malware classification in the presence of concept drift. In *SP*.
[23] Arjun Nitin Bhagoji, Daniel Cullina, Chawin Sitawarin, and Prateek Mittal. 2018. Enhancing robustness of machine learning systems via data transformations. In *CISS*.
[24] Haipeng Cai. 2020. Assessing and improving malware detection sustainability through app evolution studies. *TOSEM* (2020).
[25] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *SP*.
[26] 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).
[27] Ngoc-Tu Chau and Souhwan Jung. 2018. Dynamic analysis with Android container: Challenges and opportunities. *Digital Investigation* (2018).
[28] 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*.
[29] 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).
[30] Yizheng Chen, Zhoujie Ding, and David Wagner. 2023. Continuous Learning for Android Malware Detection. *arXiv preprint arXiv:2302.04332* (2023).
[31] Francisco Handrick da Costa, Ismael Medeiros, Thales Menezes, João Victor da Silva, Ingrid Lorraine da Silva, Rodrigo Bonifácio, Krishna Narasimhan, and Márcio Ribeiro. 2022. Exploring the use of static and dynamic analysis to improve the performance of the mining sandbox approach for android malware identification. *Journal of Systems and Software* (2022).
[32] 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*.
[33] Yuxin Ding, Xiao Zhang, Jieke Hu, and Wenting Xu. 2020. Android malware detection method based on bytecode image. *AIHC* (2020).
[34] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. On lightweight mobile phone application certification. In *CCS*.
[35] 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*.
[36] 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).
[37] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android permissions demystified. In *CCS*.
[38] 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*.
[39] 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).
[40] Han Gao, Shaoyin Cheng, and Weiming Zhang. 2021. GDroid: Android malware detection and classification with graph convolutional network. *Computers & Security* (2021).
[41] Joshua Garcia, Mahmoud Hammad, and Sam Malek. 2018. Lightweight, obfuscation-resilient detection and family identification of android malware. *TOSEM* (2018).
[42] Ross Girshick. 2015. Fast r-cnn. In *ICCV*.
[43] 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*.
[44] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. In *ICLR*.
[45] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2017. Adversarial examples for malware detection. In *ESORICS*.
[46] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask r-cnn. In *CVPR*.
[47] Ke He and Dong-Seong Kim. 2019. Malware detection with malware images using deep learning techniques. In *TrustCom*.
[48] 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*.
[49] Yiling He, Yiping Liu, Lei Wu, Ziqi Yang, Kui Ren, and Zhan Qin. 2022. MsDroid: Identifying Malicious Snippets for Android Malware Detection. In *TDSC*.
[50] Geoffrey Hinton. 2009. Deep belief networks. *Scholarpedia* (2009).
[51] 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*.

[52] TonTon Hsien-De Huang and Hung-Yu Kao. 2018. R2-d2: Color-inspired convolutional neural network cnn-based android malware detections. In *BigData*.

[53] 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*.

[54] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. 2018. Black-box adversarial attacks with limited queries and information. In *ICML*.

[55] 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*.

[56] ElMouatez Billah Karbab and Mourad Debbabi. 2021. Petadroid: adaptive android malware detection using deep learning. In *DIMVA*.

[57] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab, and Djedjiga Mouheb. 2018. MalDozer: Automatic framework for android malware detection using deep learning. *Digital Investigation* (2018).

[58] 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. In *TIFS*.

[59] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[60] Tao Lei, Zhan Qin, Zhibo Wang, Qi Li, and Dengpan Ye. 2019. EveDroid: Event-aware Android malware detection against model degrading for IoT devices. *IoTJ* (2019).

[61] 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*.

[62] Heng Li, ShiYao Zhou, Wei Yuan, Jiahuan Li, and Henry Leung. 2019. Adversarial-example attacks toward android malware detection system. *IEEE Systems Journal* (2019).

[63] Heng Li, Shiyao Zhou, Wei Yuan, Xiapu Luo, Cuiying Gao, and Shuiyan Chen. 2021. Robust android malware detection against adversarial example attacks. In *WWW*.

[64] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* (2017).

[65] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. Palmtree: Learning an assembly language model for instruction embedding. In *CCS*.

[66] Yuping Li, Jiyong Jang, Xin Hu, and Xinming Ou. 2017. Android malware clustering through malicious payload mining. In *Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings*.

[67] 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).

[68] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Deep learning for android malware defenses: a systematic literature review. *JACM* (2022).

[69] 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*.

[70] Alejandro Martín, Félix Fuentes-Hurtado, Valery Naranjo, and David Camacho. 2017. Evolving deep neural networks architectures for android malware classification. In *CEC*.

[71] 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*.

[72] Larry R Medsker and LC Jain. 2001. Recurrent neural networks. *Design and Applications* (2001).

[73] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *NIPS*.

[74] 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*.

[75] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. 2017. graph2vec: Learning distributed representations of graphs. *arXiv:1707.05005* (2017).

[76] Annamalai Narayanan, Liu Yang, Lihui Chen, and Liu Jinliang. 2016. Adaptive and scalable android malware detection through online learning. In *IJCNN*.

[77] 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*.

[78] Naser Peiravian and Xingquan Zhu. 2013. Machine learning for android malware detection using permission and api calls. In *ICTAI*.

[79] Abdurrahman Pektaş and Tankut Acarman. 2020. Deep learning for effective Android malware detection using API call graph embeddings. *Soft Computing* (2020).

[80] Abdurrahman Pektaş and Tankut Acarman. 2020. Learning to detect Android malware via opcode sequences. *Neurocomputing* (2020).

[81] 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*.

[82] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *KDD*.

[83] Junyang Qiu, Jun Zhang, Wei Luo, Lei Pan, Surya Nepal, and Yang Xiang. 2020. A survey of android malware detection with deep neural models. *CSUR* (2020).

[84] Xin Su, Dafang Zhang, Wenjia Li, and Kai Zhao. 2016. A deep learning approach to android malware feature learning and detection. In *Trustcom/BigDataSE*.

[85] 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 *SAC*.

[86] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. 2017. Revisiting unreasonable effectiveness of data in deep learning era. In *CVPR*.

[87] Lichao Sun, Zhiqiang Li, Qiben Yan, Witawas Srisa-an, and Yu Pan. 2016. SigPID: significant permission identification for android malware detection. In *MALWARE*.

[88] Mingshen Sun, Tao Wei, and John CS Lui. 2016. Taintart: A practical multi-level information-flow tracking system for android runtime. In *CCS*.

[89] 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).

[90] Zhaonan Sun, Nawanol Ampornpunt, Manik Varma, and Svn Vishwanathan. 2010. Multiple kernel learning and the SMO algorithm. In *NIPS*.

[91] Fnu Suya, Jianfeng Chi, David Evans, and Yuan Tian. 2020. Hybrid batch attacks: Finding black-box adversarial examples with limited queries. In *Security*.

[92] 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).

[93] R Vinayakumar, KP Soman, and Prabaharan Poornachandran. 2017. Deep android malware detection and classification. In *ICACCI*.

[94] Ji Wang, Qi Jing, Jianbo Gao, and Xuanwei Qiu. 2020. SEdroid: A robust Android malware detector using selective ensemble learning. In *WCNC*.

[95] Wei Wang, Mengxue Zhao, and Jigang Wang. 2019. Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. *AIHC* (2019).

[96] 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. In *TOSEM*.

[97] 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).

[98] 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*.

[99] Yueming Wu, Deqing Zou, Wei Yang, Xiang Li, and Hai Jin. 2021. HomDroid: detecting Android covert malware by social-network homophily analysis. In *ISSTA*.

[100] Xusheng Xiao and Shao Yang. 2019. An image-inspired and cnn-based android malware detection approach. In *ASE*.

[101] 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. In *TDSC*.

[102] Ke Xu, Yingjiu Li, Robert Deng, Kai Chen, and Jiayun Xu. 2019. Droidevolver: Self-evolving android malware detection system. In *EuroS&P*.

[103] Ke Xu, Yingjiu Li, and Robert H Deng. 2016. Iccdetector: Icc-based malware detection on android. *TIFS* (2016).

[104] 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*.

[105] Jinpei Yan, Yong Qi, and Qifan Rao. 2018. LSTM-based hierarchical denoising network for Android malware detection. *Security and Communication Networks* (2018).

[106] 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*.

[107] 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*.

[108] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. 2014. Droid-sec: deep learning in android malware detection. In *SIGCOMM*.

[109] Peter Zegzhda, Dmitry Zegzhda, Evgeny Pavlenko, and Gleb Ignatev. 2018. Applying deep learning techniques for Android malware detection. In *ICSIN*.

Table 10: 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 we present 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[18]	NDSS	2014	●	●	○	○	✔	✘	5,560	123,453	94%	**1%**	87%
MamaDroid[69]	NDSS	2017	○	●	○	○	✔	✘	35,493	8,447	97%	2%	96%
Mclaughlin et al.[71]	CODASPY	2017	○	●	○	○	✘	✔	13,637	13,758	95%	**1%**	97%
HinDroid[51]	KDD	2017	○	●	○	○	✔	✘	1,216	1,118	**99%**	2%	**99%**
DeepRefiner[104]	EuroS&P	2018	●	●	●	○	✘	✔	62,915	47,525	98%	2%	98%
Kim et al.[58]	TIFS	2018	●	●	○	●	✔	✔	21,260	20,000	**99%**	**1%**	**99%**
MalScan[98]	ASE	2019	○	●	○	○	✔	✘	15,430	15,285	—	—	98%
SDAC[101]	TDSC	2020	○	●	○	○	✔	✔	34,497	35,645	98%	**1%**	**99%**
HomDroid[99]	ISSTA	2021	○	●	○	○	✔	✘	3,358	4,840	97%	4%	95%
Xmal[96]	TOSEM	2021	●	●	○	○	✔	✘	15,570	20,120	98%	2%	98%
RAMDA[63]	WWW	2021	●	●	○	○	✔	✘	21,621	36,862	93%	**1%**	90%
MSDroid[49]	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.

[110] 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*.

[111] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *ESEC/FSE*.

[112] 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).

[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 *ISCC*.

[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 *MSWiM*.

[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. *TKDE* (2021).

A APPENDIX

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 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 standardized techniques across all tasks. For feature extraction, we use Androguard [1] to disassemble APK files and derive features such as permissions, intents, and program graphs. LibRadar [9] 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 ML models, the scikit-learn library is our choice for traditional ML algorithms like SVM, KNN, and RF. On the other hand, for DL architectures such as CNN, GNN, and AE, we resort to Pytorch [10]. This uniform approach ensures a balanced evaluation, concentrating purely on the uniqueness and performance of each method.

A.3 Reproduction of Selected Approaches

Table 10 offers a summary of the 12 representative approaches we have selected from leading publications in security [18, 49, 58, 69, 71, 101, 104], software engineering [96, 98, 99], and machine learning [51, 63]. 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 and comparison, we provide the hyper-parameters of these approaches.

Drebin. We replicate Drebin [18] utilizing a linear SVM with $C = 1$, and set the maximum number of iterations to 1000.

MamaDroid. MamaDroid [69] has two abstract mode *i.e.*, package and family. In our experiments, we choose 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. [71] 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 [51], 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 [90], applying the default settings for the hyper-parameters.

DeepRefiner. As discussed in Section 3.2, DeepRefiner [104] 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. [58] 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 [98] is replicated with a KNN classifier, with the number of neighbors set to 3.

SDAC. SDAC [101] 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 [99] is executed using a KNN classifier, with the number of neighbors set to 1.

Xmal. Xmal [96] 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 [63]'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 [49] 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.

Algorithm 1: Malware Evolution Evaluation Algorithm

Function SingleEvaluation
  Input: Data of Year $\mathcal{D}_y$, Data of N Months $\{\mathcal{D}_{M_1}, ..., \mathcal{D}_{M_N}\}$, Metric f
  Output: $AUT(f,0)$, $AUT(f,N)$
  /* Partition $\mathcal{D}_y$ into training, validation, and test according to 8:1:1 */
  $\mathcal{D}_{train}, \mathcal{D}_{val}, \mathcal{D}_{test} \leftarrow$ PartitionData (D_y)
  Empty List $L \leftarrow [\,]$
  Best Model $M_y \leftarrow$ TrainModel (D_{train}, D_{val})
  /* Testing on the current year data */
  $AUT(f,0) \leftarrow$ TestModel $(M_y, \mathcal{D}_{test})$
  /* Testing on the N months data */
  if $\{\mathcal{D}_{M_1}, ..., ..., \mathcal{D}_{M_N}\} \neq \{\phi\}$ **then**
    foreach *Month Data* $D \in \{\mathcal{D}_{M_1}, ..., \mathcal{D}_{M_N}\}$ **do**
      $L \leftarrow L +$ TestModel (M_y, D)
    end foreach
    $AUT(f,N) \leftarrow$ CalculateAUT (f, L, N)
  end if
  else
    $AUT(f,N) \leftarrow$ NONE
  end if

Function RollingEvaluation
  Input: Data of M Years $\{\mathcal{D}_{Y_1}, ..., \mathcal{D}_{Y_M}\}$, Time Decay N, Metric f
  Output: $AVG\_AUT(f,0)$, $AVG\_AUT(f,N)$
  Empty List $L_{AUT_0} \leftarrow [\,]$
  Empty List $L_{AUT_N} \leftarrow [\,]$
  foreach *Year Data* $\mathcal{D}_{Y_i}$ *in* $\{\mathcal{D}_{Y_1}, ..., \mathcal{D}_{Y_M}\}$ **do**
    /* Select N months data */
    Month List $L_m \leftarrow$ SortDataInMonths $(\mathcal{D}_{Y_{i+1}}, ..., \mathcal{D}_{Y_M})$
    if *Length* $(L_m \geq N)$ **then**
      $AUT(f,0), AUT(f,N) \leftarrow$
      SingleEvaluation $(\mathcal{D}_{y_j}, L_m[0:N], f)$
      $L_{AUT_0} \leftarrow L_{AUT_0} + AUT(f,0)$
      $L_{AUT_N} \leftarrow L_{AUT_N} + AUT(f,N)$
    end if
    else
      $AUT(f,0), \_ \leftarrow$
      SingleEvaluation $(\mathcal{D}_{y_j}, \{\phi\}, f)$
      $L_{AUT_0} \leftarrow L_{AUT_0} + AUT(f,0)$
    end if
  end foreach
  /* In our experiments, we set N as 3, 6, 9, ..., 24 */
  $AVG\_AUT(f,0) \leftarrow$ SUM (L_{AUT_0}) / Length(L_{AUT_0})
  $AVG\_AUT(f,N) \leftarrow$ SUM (L_{AUT_N}) / Length(L_{AUT_N})

Figure 8: The rolling algorithm for evaluating the Android malware evolution.

A.4 AUT, ASR, and APR

In assessing a classifier's resilience against temporal degradation, we employ the Area Under Time (AUT) metric as suggested by [81]. 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 [61]. They are

Table 11: 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%)

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 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.5 Malware Evolution Algorithm

To evaluate the evolution of Android malware, we propose a rolling algorithm, as illustrated in Figure 8. In our experiment, we use the data from 2011 to 2020, and the malware evolution periods are set as 3, 6, 9, 12, 15, 18, 21, and 24 months.

A.6 Additional Results

Table 12 presents the detailed results of training data size analysis. In Sec. 4.3, 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.

Table 12: The effectiveness of the selected approaches using different size 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%)

For the analysis of malware evolution, the results of AUT(TPR, N) and AUT(FPR, N) are reported in Table 11 (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.