Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 09, No. 04, December 2023.

LONGCGDROID: ANDROID MALWARE DETECTION THROUGH LONGITUDINAL STUDY FOR MACHINE LEARNING AND DEEP LEARNING

Abdelhak Mesbah[1]*, Ibtihel Baddari[1] and Mohamed Amine Riahla[2]

(Received: 30-Aug.-2023, Revised: 3-Oct.-2023 and 24-Oct.-2023, Accepted: 31-Oct.-2023)

ABSTRACT

This study aims to compare the longitudinal performance between machine-learning and deep-learning classifiers for Android malware detection, employing different levels of feature abstraction. Using a dataset of 200k Android apps labeled by date within a 10-year range (2013-2022), we propose the LongCGDroid, an image-based effective approach for Android malware detection. We use the semantic Call Graph API representation that is derived from the Control Flow Graph and Data Flow Graph to extract abstracted API calls. Thus, we evaluate the longitudinal performance of LongCGDroid against API changes. Different models are used; machine-learning models (LR, RF, KNN, SVM) and deep-learning models (CNN, RNN). Empirical experiments demonstrate a progressive decline in performance for all classifiers when evaluated on samples from later periods. However, the deep-learning CNN model under the class abstraction maintains a certain stability over time. In comparison with eight state-of-the-art approaches, LongCGDroid achieves higher accuracy.

KEYWORDS

Android security, Malware detection, Machine learning, Adjacency matrix, Longitudinal evaluation.

1. INTRODUCTION

The ever-expanding landscape of mobile applications has brought about new challenges in ensuring the security and privacy of users' devices. Among these challenges. Android malware stands out as a persistent and evolving threat, requiring robust and effective detection mechanisms. With a market share near 72% as of the first quarter of 2023 [1], the Android platform leads the mobile OS. Its popularity, accentuated by the extensive availability of diverse third-party Android app-distribution channels, makes it a prime target for malware. It is estimated that more than 5 million Android malware samples have been seen in the wild as of the first quarter of 2023 [2].

Traditional signature-based methods have proven insufficient in keeping pace with the constantly mutating nature of malware, necessitating the exploration of advanced techniques. Aware of this fact, researchers use machine-learning (ML) and deep-learning (DL) techniques to develop malware analysis and detection systems by analyzing the behavior of the apps and extracting the set of features that best describe their behavior based on dynamic and/or static features [3]. Dynamic analysis refers to a method of analyzing the behavior of apps in real time [4], while they are running in a controlled environment, known as a sandbox. This approach involves monitoring the apps' interactions with the operating system, hardware and other apps, simulating various user inputs and system events to understand how the apps behave under different conditions. Dynamic analysis is different from static analysis, inspecting the code and data of apps without executing them [5]. Instead, it involves examining the contents of the apps, which include the app's code, resources and manifest file, among others. This analysis is performed before the app is run, giving an overview of the app's structure, potential vulnerabilities and behavior without actually executing it on a device or emulator. Both analysis approaches are complementary and can be combined for Android malware detection.

Many works and tools have been developed for malware detection, involving various analysis methods and feature usages. However, most studies present performance results using

1. A. Mesbah (Corresponding Author) and I. Baddari are with Department of Computer Science, Faculty of Sciences, LIMOSE Laboratory, University M'Hamed Bougara of Boumerdes, Algeria. Email: abdelhak.mesbah@univ-boumerdes.dz
2. M. A. Riahla is with Department of Electrical Systems Engineering, Faculty of Technology, LIMOSE Laboratory, University M'Hamed Bougara of Boumerdes, Algeria.

conventional methods without considering the period or age of the test samples used during the extraction and evaluation. As apps evolve, their characteristics and behavior may change, which can affect the accuracy and sustainability of the models. A few existing works focus on characterizing the behavior of Android apps in relation to the temporal evolution of data, specifically addressing the phenomena commonly referred to as concept drift and data drift. These studies propose methodologies that enable adaptation to changes in the data and aim to mitigate the negative consequences that may arise over time. On the one hand, for security reasons, but not necessarily for Android malware detection, [6]-[8] discussed comprehensive approaches towards understanding and adapting to the rapid evolutionary dynamics of the mobile app ecosystem, mainly focusing on Android, to improve app quality, security and usability. The authors proposed a continuous ecosystem mining and characterization approaches to systematically study and understand the evolutionary dynamics of the Android ecosystem. On the other hand, different studies contributed to a deeper understanding of the intricacies involved in malware classification and the importance of adapting to the evolutionary dynamics of the Android ecosystem to improve malware-detection accuracy and reliability. In [9]-[13], the authors underscored the importance of a comprehensive approach to understanding app evolution in enhancing malware classification. As the evolution problem is multi-faceted, it encompasses the continuous adaptation of malware to evade detection, the evolution of goodware to stay secure and functional and the changes in the Android platform that might affect both malware and goodware behaviors. The studies demonstrated that a more accurate and reliable malware classification can be achieved by eliminating experimental bias, addressing platform fragmentation and analyzing malware behavior over time.

The features used for malware detection may exhibit dominance or effectiveness in certain years, but not in others, which highlights the importance of considering the temporal aspect and continuously evaluating the effectiveness of the chosen features and models longitudinally. Hence, in this study, a longitudinal performance comparison is made between machine-learning and deep-learning classifiers, through the proposed LongCGDroid, an image-based, fully automated process to detect Android malware using static and pseudo-dynamic analysis. LongCGDroid uses a classic representation of programs in program-analysis techniques; namely, the Call Graph API, which is a representation of the interactions between various functions within a software application. It is used to visualize and analyze the flow of function calls made by an Android app during its execution. This call graph is constructed based on two main steps: the extraction of the Control Flow Graph (CFG) to reflect what a program intends to behave, as well as how it behaves (i.e., possible execution paths), such that malware behavior patterns can be captured easily and the Data Flow Graph (DFG) that represents the data dependencies between several operations, in our case links between API calls. Both CFG and DFG are extracted through static and pseudo-dynamic analysis on the instruction level (i.e., smali files in the Dalvik executions). By analyzing each call graph's semantics, we construct an abstracted 2-D adjacency matrix for each app that represents the relation between the abstracted API's calls.

The contributions of this study can be summarized as follows:

- Generating a large date-labeled dataset containing 200K Android apps (100k malware and 100k goodware) with a 10-year range from 2013 to 2022; for each year, 20K Android applications are used (10k malware and 10k goodware).

- Proposing and developing LongCGDroid, a new malware-detection image-based system, using the semantic of the call graph API with different modes of abstraction. All execution paths in terms of the invoked APIs are captured through the DFG. The pseudo-dynamic analysis is applied; i.e., apps are not executed; instead, all execution paths are analyzed *via* the DFG.

- We propose the use of different API call abstractions to represent them in the adjacency matrices; i.e., abstract API calls to either the method name (e.g. *java.io.File.getPath*), the class name (e.g. *java.io.File*) or its package name (*java.io*). Abstraction provides resilience to API changes in the Android framework as classes and packages are added and removed less frequently than single-method API calls. We evaluate the classifiers with each abstraction, as the LongCGDroid can operate with each mode.

- We propose the use of an adjacency matrix to effectively engineer Android APIs for machine-learning tasks. This approach allows better feature embedding. By only using the most popular

features, the graph's size is significantly reduced. Therefore, the adjacency matrix is also greatly simplified with an inferred size for each level of abstraction: 100×100 for package abstraction, 200×200 for class abstraction and 300×300 for method abstraction, allowing a fast data-processing stage.

- We investigate the extent of performance decay over time for various machine-learning and deep-learning classifiers trained with features extracted from date-labeled goodware and malware samples. The classifiers are then tested on goodware and malware samples from a later time period, thus mimicking a true zero-day scenario that gives a more realistic view of performance than the traditional evaluation approach.
- We conduct comparative experiments with eight state-of-the-art techniques [14]-[21], which fully demonstrate the effectiveness of our approach.

The paper is organized as follows. We begin by providing a background as contextual foundation through outlining the key concepts in Section 2, followed by a discussion of the related works on malware detection in Section 3. The LongCGDroid framework and our methodology are introduced in Section 4. LongCGDroid's evaluation and comparative analyses are presented in Sections 5. Conclusions are presented in Section 6.

2. BACKGROUND

2.1 Android APK Format

Android Application Package (APK) is the file format used to distribute and install applications on Android devices [22]. It is a compressed archive file that contains all the necessary components and resources required to run an Android application. The components of an APK include the *Manifest File*, which contains essential information about the application, such as its package name, version code and permissions. Compiled code *classes.dex* is represented as Dalvik Executable (DEX) files. These files contain the bytecode generated from the application's Java source code. The Resources folder contains files, such as images, layouts and other resource files required for the user interface and functionality. Libraries are required by the application to interact with the underlying hardware or perform certain platform-specific tasks.

2.2 Static Analysis

Static analysis is one of the most commonly used techniques in malware detection, focusing on inspecting the APK without executing the code [5]. By examining the APK's structure and contents, static analysis extracts valuable insights to assess potential threats. This analysis method is advantageous, as it allows for a rapid examination of large datasets, providing a preliminary screening of applications. During static analysis, numerous static features are extracted from the APK [3]. These features encompass various characteristics of the application, enabling the classifier to distinguish between benign and malicious samples. Some common static features include permissions, API calls, string analysis and code structure. The information extracted through static analysis is then converted into a standardized set of explanatory features that are later processed by machine-learning algorithms. To carry out static analysis on APK files, specialized tools like Androguard [23] can be employed to access fields and components declared within the manifest file. It also facilitates the construction of various graph structures representing the code's execution flow. Notably, two essential graph types are Call Graphs and CFGs. Call Graphs are built by tracing call instructions in the code, while CFGs encompass conditional and loop statements (if, switch, for, while, …etc.), reflecting the code's jumps. Two main tools can be used to extract call graphs or CFGs: Androguard and Flowdroid [24].

2.3 Pseudo-dynamic Analysis

Pseudo-dynamic program-flow analysis is a technique used in software analysis to understand the potential execution paths of a program without executing it in a real runtime environment [25]. Unlike traditional dynamic analysis, which involves running the program on an actual system, pseudo-dynamic program-flow analysis performs a form of simulated execution with code instrumentation. In this approach, one can track and log the control flow as the code executes. These additional instructions act as probes or logging points, allowing researchers to observe the program's

execution path and capture critical information about the program's behavior. Pseudo-dynamic program-flow analysis offers several advantages over traditional static analysis. It enables researchers to gain insights into how the program would behave under certain conditions without the need to execute it in a real environment. Additionally, it can help identify potential security vulnerabilities, resource bottlenecks or unintended program paths. Particularly, all execution paths can be captured in terms of the operation codes, so-called opcodes.

3. RELATED WORKS

Android malware detection is a well-studied area in the information-security literature. There have been several works that focus on machine learning-based detection. Despite this, very few of them consider the investigation of long-term performance and a longitudinal resilience evaluation. In this section, we classify them into two categories: longitudinal-based and conventional-based evaluations.

3.1 Longitudinal-based Machine-learning and Deep-learning Evaluation

The majority of existing research on Android malware detection does not address longitudinal resilience or long-term performance issues. In [26], the paper presents a dedicated longitudinal study of the performance of machine-learning classifiers for Android malware detection aiming for a sustainable system. The study is undertaken using very basic features, such as API calls, permissions and intents extracted from Android apps. It takes into consideration a dataset constructed from apps first seen between 2012 and 2016. The aim is to investigate the extent of performance decay over time for various machine-learning classifiers, such as Support Vector Machines (SVMs), Naïve Bayes (NB), Random Forest (RF), Simple Logistic (SL) and Decision Tree (DT). The SL was the most resilient over time. In [14], MAMADROID is presented as an Android malware detection system that relies on app behavior. It builds a behavioral model in the form of a Markov chain from the API call sequences, which is used to extract features and perform classification. The paper includes a longitudinal evaluation of the accuracy using a dataset of 8.5K benign and 35.5K malicious apps, collected over a period of six years, first seen between 2010 and 2016. The reported F-measure is up to 99% with a progressive diminishing performance over time, reaching average F-measure values of 86% and 75% one and two years after training, respectively. Maldozer [15] is a deep learning-based framework that relies on raw API call sequences for identifying malware apps. To evaluate their tool, they constructed a dataset of 33k goodware and 38 malware apps. It is also longitudinally evaluated on apps collected from four consecutive years, 2013 to 2016. The authors achieved 96% to 99% detection accuracy. Similarly, RevealDroid [19] addresses the increasing threat of Android malware and the limitations of existing detection techniques, particularly their inability to handle certain obfuscations and scalability issues. The features selected for RevealDroid focus on categorized Android API usage, reflection-based features and features from native binaries of apps. In [16], the authors delve into the evolving nature of malware and its implications on detection strategies, particularly emphasizing the phenomenon of concept drift. Through their system named DroidEvolver, they underscore the continuous evolution of malware as a significant challenge to cybersecurity, which in turn affects the performance of machine-learning models employed in malware detection. The approach is designed to automatically and continually update itself during malware detection without human intervention or retraining with accurate labels. DroidCat [20] introduces a dynamic app classification technique for Android malware detection and categorization. The system uses dynamic features based on method calls and inter-component communication (ICC) intents. The authors assessed DroidCat's performance across the spanning of nine years, with a relatively small dataset of 34343 samples collected from 2009 to 2017. In [21], the author presents DroidSpan, a dynamic-analysis system based on a behavioral profile for Android apps. This system captures sensitive access distribution from lightweight profiling during runtime. The system's dynamic approach focuses on the extent and distribution of exercised sensitive accesses and vulnerable method-level control flows in app executions. By leveraging this dynamic behavioral profile, DroidSpan aims to offer sustainability in malware detection, especially in the context of the evolving Android platform and its applications. Through all the longitudinal experiments made by the author over the years, the system does not exceed an F1-score of 91%.

All of these existing Android malware-detection studies have employed a relatively small dataset containing a few thousand apps. Moreover, the evaluation timelines of these studies did

not extend beyond 2018. Furthermore, none of the studies provided a comparison of performance between machine- learning and deep-learning models.

3.2 Conventional-based Machine-learning and Deep-learning Evaluation

Early research on Android malware detection predominantly employed traditional machine-learning algorithms. Most machine learning-based malware techniqes still use features such as Android app permissions, API calls and control flow graphs to distinguish between benign and malicious apps.

For instance, Jung et al. [27] proposed a malware detection approach based on the frequency of API calls. The authors' main idea is based on the construction of two ranked lists of popular Android API calls: a benign API call list and a malware API call list. The RF classifier is applied to a dataset of 60,243 apps (30,159 goodware and 30,084 malware) using each list as a feature. The evaluation shows that the classifier achieves promising results. DroidSieve [17] is introduced as a malware-classification system that employs obfuscation-resilient static analysis of Android apps. The authors delve into the challenges posed by obfuscation in Android malware, discussing various obfuscation techniques and their impacts on static analysis. They propose a diverse set of features for robust classification, covering both non-obfuscated and obfuscated malware and offering a high level of accuracy and efficiency while addressing the challenges posed by obfuscation in malware analysis and detection. In [28]-[29] the authors presented frameworks that combine permission and API calls to detect malicious Android apps using machine-learning methods, extracting permissions from each app's profile information and APIs from the APK to represent API calls and validating the algorithm's performance through experiments on real-world apps with a small dataset. Similarly, [30]-[33] proposed an intelligent model for detecting malware applications using machine-learning algorithms. The frameworks are based on static malware analysis to extract features, such as permissions and API calls using multi-level feature selection algorithms.

Graph-based approaches have gained traction over the years, leveraging the structural and behavioral characteristics of Android apps to enhance detection accuracy. DroidMiner [34] is a system that uses static analysis to automatically mine malicious program logic from known Android malware and seeks out these threat-modality patterns in other unknown Android apps. They also describe a two-tiered behavioral graph model, control-flow graphs and call graphs, for characterizing Android application behavior and labeling its logical paths within known malicious apps as malicious modalities. In [35], the paper proposes a root exploit malware recognition system called DroidExec, which reduces the impact of wide variability in Android malware detection. The system uses a Bipartite Graph Conceptual Matching of graph edit distance to fold redundant function-relation graphs and conceptually cripple wide variability. DroidOL [36] and DeepCarta [37] propose frameworks that capture security-sensitive behaviors from apps in the form of structural information captured through graphs.

With the rise of deep learning, researchers started to leverage its capabilities to achieve more robust and accurate malware detection. EveDroid [38] uses event groups to describe app behaviors in event level and function clusters to represent behaviors in each event. A neural network is used to aggregate multiple events and mine the semantic relationship among them. [39] proposes a multimodal learning approach for Android malware detection using deep-learning techniques. It uses static analysis to extract permission and hardware features. In [40]-[42], the authors propose systems that use deep convolutional neural networks to learn from opcode sequences. The systems extract raw opcode sequences from decompiled Android files and train the network to effectively learn feature information and accurately detect malicious programs. R2-D2 [43] is system that converts the bytecode of *classes.dex* from APK to RGB color codes and stores them as fixed-size color images. These images are then input to the CNN for automatic feature extraction and training. Similarly, Vu et al. [18] propose AdMat, a grayscale image-based approach to treat malware detection. The authors construct an adjacency matrix for each app, serving as input images to the CNN model. DroidDivesDeep [44] proposes a method for classifying Android malware using low-level monitorable features and deep neural networks.

In our study, we aim to compare machine-learning and deep-learning classifiers by addressing the longitudinal aspect, which offers a more authentic assessment of effectiveness compared to

conventional evaluation methodologies that overlook the temporal aspect of app appearances. We evaluate our framework LongCGDroid using a large dataset of date-labeled apps with a specific focus on utilizing graph-based features' approach, where both control and data-flow graphs are considered.

4. METHODOLOGY

The proposed LongCGDroid consists of five consecutive main stages. Figure 1 shows the framework of our approach. The first stage consists of collecting and cleaning the dataset. At the second stage, static and pseudo-dynamic analyses of Android apps are done, to extract API call graph, CFGs and DFGs of the samples under inspection. Particularly, all execution paths are captured using the extracted CFGs. These CFGs are used in the pseudo-dynamic analysis of the DFGs to extract all the invoked built-in APIs. Data-flow analysis is used to track the data across apps. It relies on an underlying abstract semantics of Android apps. Data flow shows the data dependencies between functions. At the third stage, we conduct a frequency analysis to select the APIs which are the most used. We further refine the API list to include only those with a usage difference higher than or equal to a certain threshold. Then, considering the abstraction mode, we select the number of APIs to use in order to construct and encode the adjacency matrices. At the fourth stage, we train the selected models using a part of the dataset. And finally, a longitudinal evaluation is made over years using the rest of the dataset.

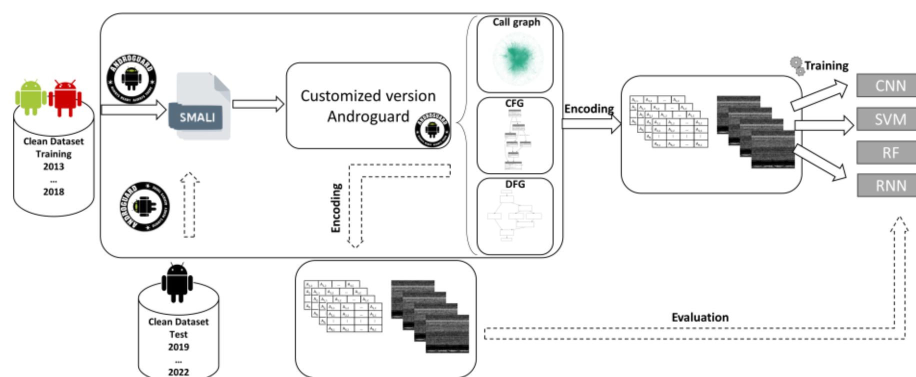

Figure 1. Process flow of LongCGDroid.

4.1 Dataset

We collect samples mainly from two datasets: Androzoo [45] for goodware apps and Virushare [46] for malware apps, as each app in these datasets is associated with the first seen timestamp, which is essential for our study. Our dataset is constructed from apps first seen between 2013 and 2022. During the collection process, we also took into consideration the apps from Maldozer [15] where we collected all the apps based on their hashes.

Initially, before cleaning, the dataset was constructed from 269970 apps; for each year, more than 20k apps are collected (> 10k goodware apps, >10k malware apps). To construct the final dataset, we considered cleaning phases (Figure 2). The first phase checks if the app is not corrupted and can be decompiled through Androguard, as some Android apps failed to decompress because of bad CRC-32 redundancy checks and errors during unpacking. Next, the app is automatically uploaded through our Python scrapper to VirusTotal [47], which is a tool that allows users to upload files, including APKs and scan them using a collection of antivirus engines. Once the app is scanned, it is labeled depending on the number of positive (malware) alerts. This number is what we call VTD (VirusTotal Detection) and it is used to relabel the samples using a threshold to establish the level of consensus required to label an APK as malware or goodware [48]. In our case, to label apps as malware, the VTD must be at least equal to 4. On the other hand, apps are labeled as goodware if the VTD is equal to 0, which means that all antiviruses marked it as clean. These verifications and labeling choices are influenced by the nature of the dataset, which includes older APKs. Although datasets, such as AndroZoo and VirusShare, which are used to create our dataset, already use VirusTotal results for application classification, these provided results may change over time. (e.g. as a result of updates to the engine, with the primary objective of enhancing detection capabilities or due to the addition or removal of

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 09, No. 04, December 2023.

engines from the platform).

The final dataset consists of 200K Android apps over a period of 10 consecutive years, from 2013 to 2022. For each year, 10k goodware and 10k malware apps are collected. To the best of our knowledge, we are leveraging the largest dataset of malware ever used in a longitudinal study on Android malware detection. For the sake of our study, it was decided to merge 7 years from 2013 to 2018 to form the training dataset, thus mimicking as best as possible the real world of the malware classifiers. Spanning 7 years ensures the use of 140k apps for training. Moreover, the apps from the 2019 to 2022 date range were intended solely to be used for evaluation during the longitudinal experiments.

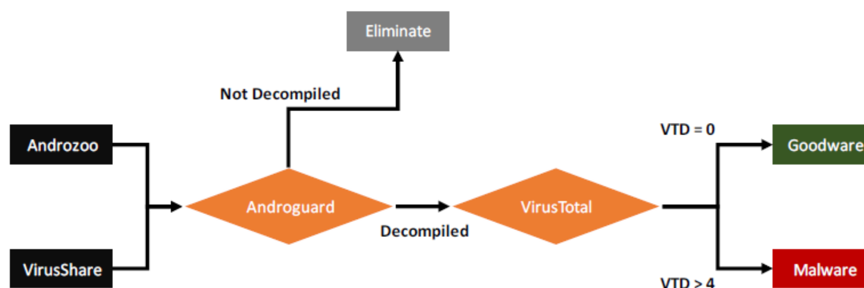

Figure 2. Dataset cleaning phases.

4.2 Feature Extraction

In this section, the entire feature-extraction process is described in detail, from the de-compilation of the APK files to the extraction of the different graphs and the construction of the adjacency matrices.

4.2.1 API Usage and Abstraction

API calls play an essential role in the operation of Android applications. They enable developers to access operating system functionalities and interact with hardware and software components. In our study, we distinguish between two types of APIs: built-in APIs (Android system APIs) and user-defined APIs (APIs defined by application developers). Exclusively in this work, we consider only the built-in APIs, as they represent consistent patterns and are available across different Android versions and devices, making them reliable indicators of app behavior. Motivated by the enhancement of the robustness of LongCGDroid, these API calls are abstracted to three different modes: package, class and method levels to cope with the changes of the Android system. The intuition behind this is that abstraction provides resilience to API changes in the Android framework, as classes and packages are added and removed less frequently than single API calls at the method level. Therefore, focusing only on specific methods can make the data less comparable and consistent over an extended period of time. For each API call extracted from the scanned app, it is processed into three abstractions, as illustrated in Figure 3. An API such as "android.telephony.SmsManager.sendTextMessage()" at the method level is abstracted to its class level as "Android.telephony.SmsManager", where the class part is preserved and to its package level as "Android.telephony", where the package part is preserved. These abstractions allow us to group together similar APIs and simplify data representation.

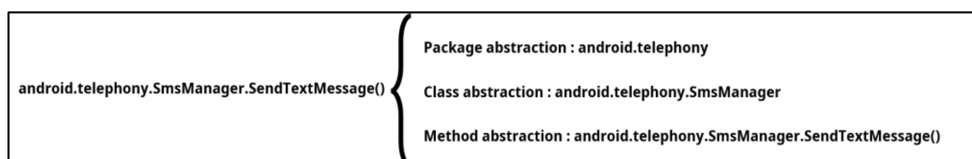

Figure 3. API abstraction modes.

4.2.2 Graph Extraction

Examining the graph call API can help effectively reveal the intention of an app. That depicts the calling relationships between different functions within a program. It shows how functions call other functions, creating a hierarchical structure of function calls. There are several tools for generating call graph APIs, such as Androguard or Flowdroid. Unfortunately, the call graph generated by these existing tools lacks inter-procedural built-in API calls. This is why LongCGDroid uses a custom

version of Androguard to extract the call graph API and generate the adjacency matrix.

To better clarify the differences between what is done by the original Androguard and our custom version, we employ a running example using a real-world malware sample. Specifically, Figure 4 lists a class extracted from the decompiled APK of malware disguised as a "*camera photo taking and editing*" (with package name *com.cp.camera*), which contains a remote app-related string. It is used to intercept SMS content for premium SMS fraud. To ease presentation, we focus on the portion of the code snippet executed in one function "loginByPost".

```java
public String loginByPost(String code) {
    String str = Build.VERSION.RELEASE;
    String str2 = Build.MODEL;
    String phoneNumber = getPhoneNumber();
    String deviceId = getDeviceId();
    try {
        HttpURLConnection urlConnection =
            (HttpURLConnection) new URL(
                "http://139.59.107.168:8088/appsharejson?code=" + code)
                .openConnection();
        urlConnection.setRequestMethod("POST");
        urlConnection.setReadTimeout(5000);
        urlConnection.setConnectTimeout(5000);
        urlConnection.setDoOutput(true);
        urlConnection.setDoInput(true);
        if (urlConnection.getResponseCode() != 200) {
            return "error";
        }
        InputStream is = urlConnection.getInputStream();
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        byte[] buffer = new byte[1024];
        while (true) {
            int len = is.read(buffer);
            if (len != -1) {
                baos.write(buffer, 0, len);
            } else {
                is.close();
                baos.close();
                return new String(baos.toByteArray());
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
        return "error";
    }
}
```

Figure 4. Malicious code (*com.cp.camera*) intercepting SMS content for premium SMS fraud.

First, the function gathers device-specific information, including the operator's name and phone number, through the *TelephoneManager* object, which gives information about the telephony services on the phone. Then, this app uploads all gathered information to the server (i.e., http://139.59.107.168:8088/). This malicious app receives the content of the message to send from the user's phone as a Json file.

The resulting sub-call graph generated by the original Androguard of the *loginByPost* function is shown in Figure 5(a). On the one hand, all the green nodes represent the user-defined APIs and the red nodes represent the built-in APIs called by the current function. On the other hand, the green arrows represent all links between user-defined APIs, the caller-callee relation, but the red nodes are not connected. No relationship is reported between built-in APIs by Androguard. This is why we use our custom Androguard version to represent the links between built-in APIs, as we can see in Figure 5(b), where red arrows are added in the sub-call graph during the pseudo-dynamic analysis to represent the caller-callee relations between built-in APIs. From this, we can infer that the

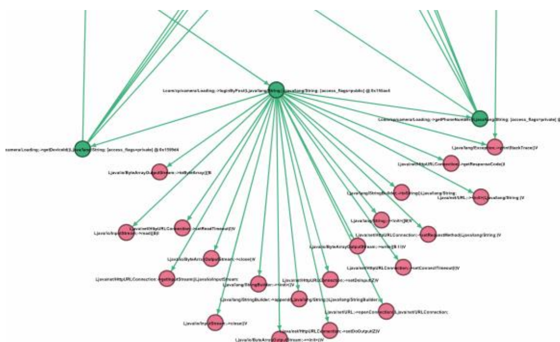

Figure 5 (a). API call graph extracted with custom original Androguard.

Figure 5 (b). API call graph extracted with Androguard highlighting external links.

loginByPost function calls the external method *java.net.url.<init>*, then the external method

java.net.url.openconnection is called and so on. These links are used to construct the adjacency matrices. This is made possible by using CFGs and DFGs to enhance the semantics of the call graph.

4.2.3 Pseudo-dynamic Analysis

We first construct a CFG for each method defined in an APK by using Androguard. We have customized the CFG extraction in order to return the graph with a Graph Modeling Language (GML) representation, which is more maintainable than the default proposed formats (i.e., png, jpg, raw). A CFG is defined as a directed graph G with the quadruple (N, E, S, F), where N is a finite set of nodes, $E \subseteq N \times N$ is a finite set of edges, where $(n, m) \in E$, if m may execute directly after n, $S \subseteq N$ is the set of starting nodes and $F \subseteq N$ is the set of exiting nodes.

We construct a CFG for each method in the call graph (Figure 6). This CFG will be used as input for the DFG. We focus on logging and tracking the built-in API calls, as they represent the features used to construct the adjacency matrix for each APK. Rather than executing the code decompiled from the c*lasses.dex* file in an emulator or sandbox, our analyzer tracks the code instruction by instruction without execution. During parsing, the analyzer uses the smali code to follow the DFG of the program instructions and register the update in a manner that mirrors a possible execution of a program, but does not compute any state information for the program.

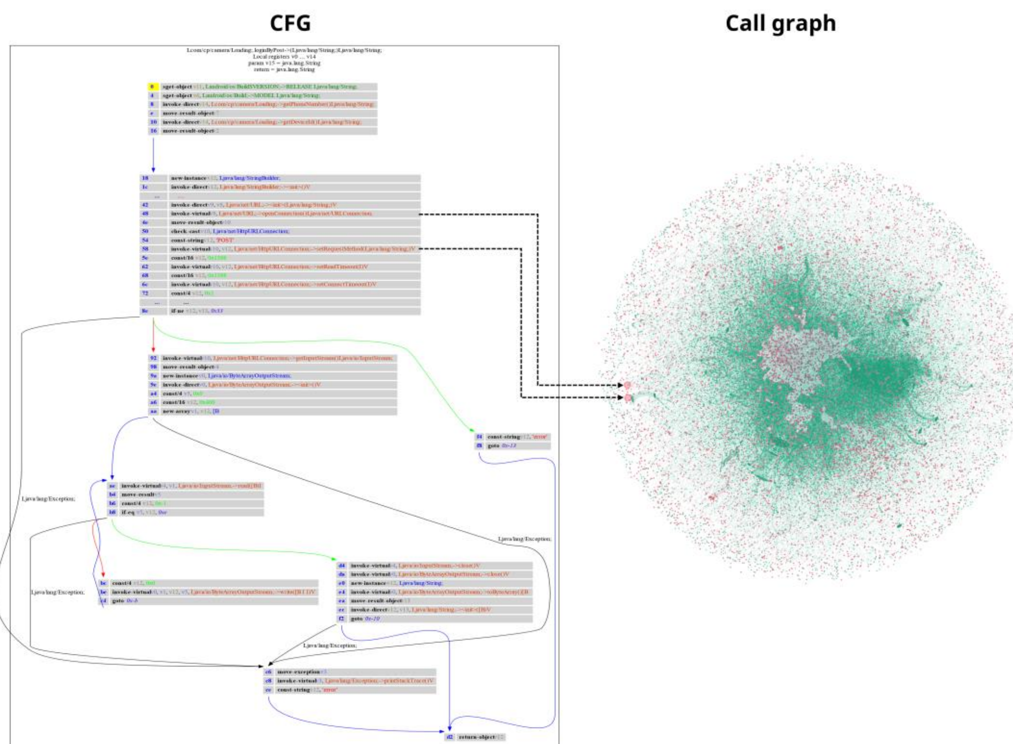

Figure 6. The CFG with smali code of the method loginByPost.

By generating the program-flow analysis through static analysis, we obtain a complete code coverage and avoid the possibility of dynamic malware obfuscation. To construct the DFG, our analyzer tracks three main categories of instructions: Regular instructions, register instructions and invoke instructions. The algorithm in Figure 7 gives the details of the extraction. This algorithm takes the CFG of a defined function f as input, represented by all possible paths and then returns the DFG of this function is represented as caller-callee relations.

Regular instructions include all instructions that do not affect the registers (e.g. *checkcast*, *instance-of*, …*etc.*); the analyzer simply steps over them with no state change, except for the update of the program counter for the length of the instruction. Register instructions are the instructions that store and manipulate data on registers during Android app execution (e.g. *move-result-object, move-object, ...etc.*). The registers' state is initialized by the methods' parameters and updated throughout the possible execution path. Invoke instructions are instructions used during an

invocation on the smali code (e.g. *invoke-virtual*, *invoke-virtual/range*, … *etc.).* They are used by LongCGDroid to extract the relation between built-in API calls as the edge of the DFG; that is, if an instruction I uses a register R, the value of which is already inferred by the register instruction as instruction J, then there is an edge from I to J.

For the sake of comprehension, we can take as an example the smali code of the function *loginByPost* in Figure 6. The instruction at the offset 0x58 is an invoke-virtual, which takes as arguments two registers, v10 and v12, and calls the function *setRequestMethod()*, as the register v10 is already inferred earlier during the analysis from the instruction 0x4e *move-result-object* and stated as the result of the instruction before, which is an invoke-virtual of the method *openConnection()*. From this, we can state that the *openConnection()* method as a caller calls the method *setRequestMethod()* as a callee. One can see this relation in the extracted call graph (Figure 6), marked by arrows with broken lines.

ALGORITHM: EXTRACT EXTERNAL API LINKS

```
     Input: CFG of the used function
     Output: Construction of the list of Caller-callee relation
1    list_callers_callees = ø
2    registers_state = initialize(method_description)
3    for each path in possible_paths
4    |    for each instruction in instructions_path
5    |    |    if instruction in register_instructions
6    |    |    |    registers_state = update_registers(instruction)
7    |    |    end if
8    |    |    if instruction in invoke_instructions
9    |    |    |    inferred_caller = get_caller(registers_state, register_number)
10   |    |    |    callee = get_callee(instruction)
11   |    |    |    list_callers_callees(inferred_caller, callee)
12   |    |    end if
13   |    end for
14   end for
```

Figure 7. Algorithm to extract built-in API links.

4.3 Graph Encoding

Once the caller-callee relations are inferred from an APK, it becomes straightforward to represent a graph as a matrix, such as the adjacency matrix. In more detail, given the list of caller-callee relations, we encode it into matrix *A* as follows: for each relation (n1; n2), we activate the element A[n1][n2] by 1, where *A* is initialized as a zero matrix. However, we have to consider three main factors:

1. **Graph Node Consistency**: This entails maintaining similarity in node values across different graphs, while also preserving equal node counts between these graphs.
2. **Matrix Size**: To ensure the uniformity of matrix sizes, the extracted nodes from each graph must exhibit consistency. Optimal matrix sizes, implying a relatively small number of nodes, are essential to facilitate swift processing and eliminate sparse matrix concerns.
3. **Feature Balance:** Balancing high-variance attributes across APK samples while avoiding matrices with inadequately activated regions is crucial. Consequently, emphasis is placed on built-in APIs to ensure consistent matrix sizes, while excluding user-defined methods.

According to the Android platform [49], there are 1081 packages, 4,853 classes and 43,800 methods in the Android API level 32. One cannot use all of these defined APIs to construct an adjacency matrix. If we take the case of packages only, the association rules that will be generated are 1167480 associations. In order to speed up the creation of adjacency matrices and avoid noisy structures, we selected different numbers of features for the different abstraction modes. For the package, class and method abstractions, we take 100, 200 and 300, respectively, from the most commonly used built-in

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 09, No. 04, December 2023.

APIs to form a matrix. The original graphs would be trimmed by the selected features, resulting in a simplified version of the adjacency matrices with the chosen sizes. This is an empirical process, as we tried different feature lengths. When the number of features for method abstraction is small (e.g., 50×50) there was a sign of learning degradation, while when the number of features is high (e.g., over 1000 × 1000), the extraction phase would take much longer. The scheme of the process was the same for all different abstraction modes. This allows a trade-off between speed and learning performance that significantly improves the recognition rate. This selection was based on a statistical study designed to identify the most representative APIs used by malware and goodware apps. Figure 8 shows the top 10-API occurrences with the highest difference in usage between malware and goodware apps.

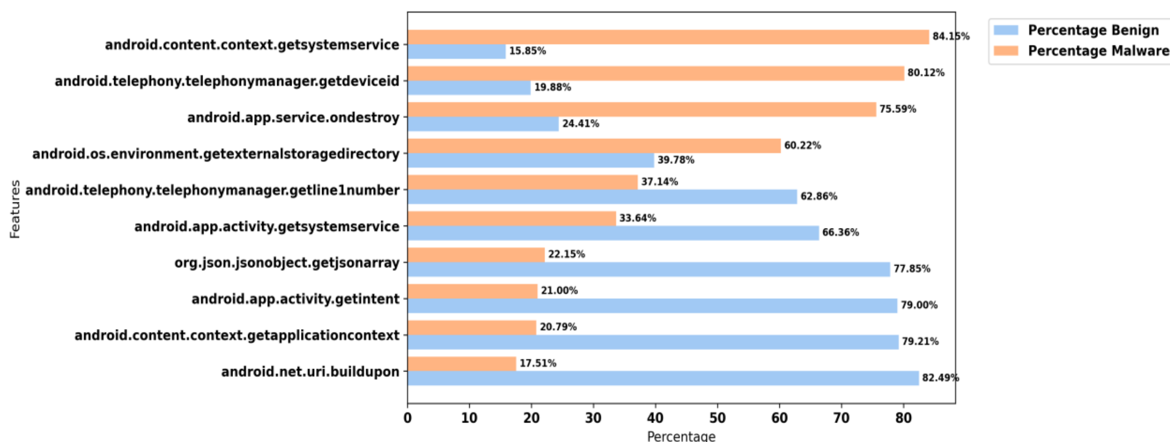

Figure 8. Frequency of the top 10 API with the highest difference between malicious and benign apps.

Once the features have been selected, the graph is converted into a 2-D adjacency matrix: the nodes represent rows and columns and if there is a connection between two nodes, the corresponding position in the matrix will be activated. As a result, each graph would form a matrix with several activated regions. Finally, we generate black-and-white images. Figure 9 presents the three different image representations of benign and malicious Android apps for each of the three abstraction modes.

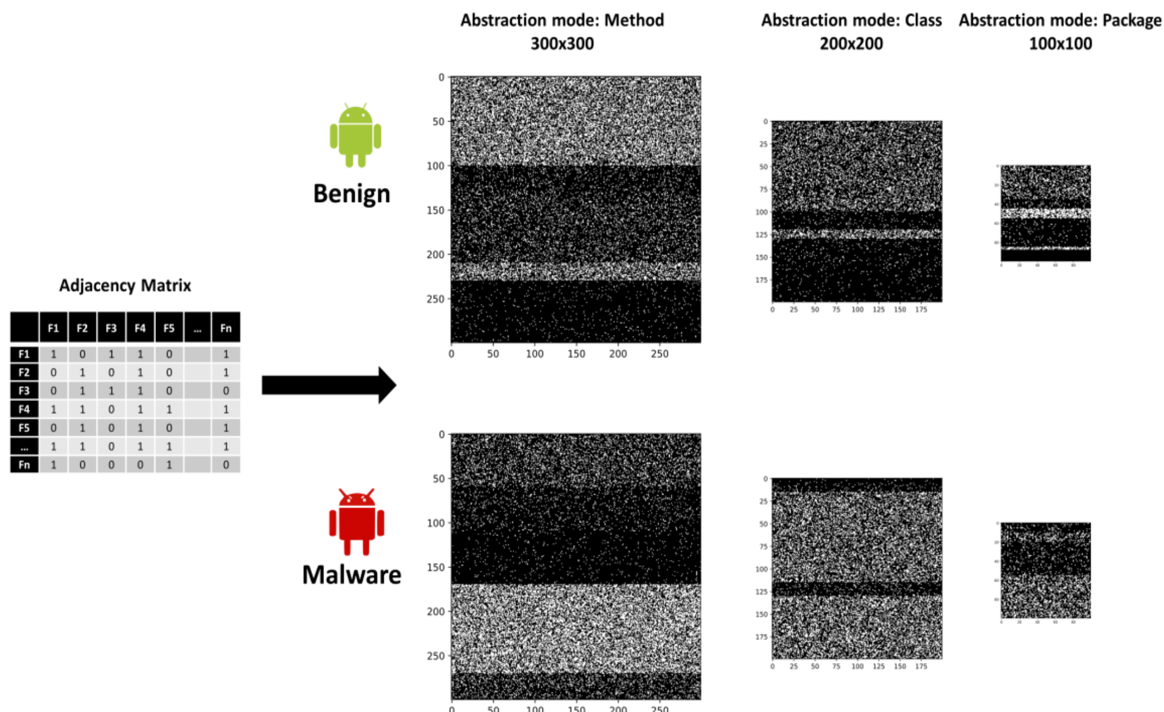

Figure 9. 2-D black and white images of benign and malicious apps in the three different abstractions.

4.4 Model Training

In this study, we compare and evaluate the longitudinal performance of machine-learning and deep-learning classifiers. To evaluate the features obtained by LongCGDroid, we applied them to the following classifiers: Machine-learning classifiers: DT, KNN, LR, RF and SVM; deep-learning classifiers: CNN and RNN-LSTM. For comparison, we also consider the API abstraction modes: method abstraction, class abstraction and package abstraction. For model construction and evaluation, the dataset is divided into training and test partitions. In order to obtain unbiased results, the test partition is always kept as a completely separate set and is never used for training or for feature engineering processes (extraction or pre-processing). The model parameters are selected using standard k-fold cross-validation (with k = 5) within the training set and following a grid search approach.

4.5 Evaluation Metrics

For this work, we considered a number of evaluation metrics that are commonly used in ML Android malware detection [3]. Note that some of these metrics, such as True Positive Rate (TPR), which is the same as Recall and False Positive Rate (FPR) or Precision, provide complementary information and should be used together to fully understand a system's performance. TPR indicates the rate of correct classification of malicious applications, while FPR indicates the rate of misclassification of clean (benign) applications. These metrics rely on: True Positives (TP), indicating the number of correct positive predictions and False Positives (FP), considering the number of incorrectly predicted negative items. We also used metrics such as accuracy and F1-score, Table 1.

Table 1. Evaluation metrics.

Metrics	Description
Accuracy	$\frac{TP + TN}{TP + TN + FP + FN}$
Precision	$\frac{TP}{TP + FP}$
TPR(Recall)	$\frac{TP}{TP + FN}$
F1-score	$\frac{2 \times Precision + Recall}{Precision + Recall}$

5. EXPERIMENTS AND RESULTS

We test the performance of the different models using behavioral-relationship features. Three series of tests are conducted in order to confirm the hypothesis that it is not enough to test a model with traditional methods, but it must also be evaluated longitudinally in order to ensure its robustness. In the first set of experiments, we analyze the accuracy of LongCGDroid by ignoring the evolution of the apps. In the second set of experiments, we conduct a longitudinal comparison of different models to evaluate their robustness. In the third set of experiments, we compare under the same conditions the LongCGDroid against eight state-of-the-art baseline detectors that rely on API calls.

During this comparison, we take into consideration the three different abstraction modes, denoted as P for package abstraction, C for class abstraction and M for method abstraction.

5.1 Experiment 1: Base Line Training Using a Temporally Imbalanced Dataset

Before analyzing the scenario related to longitudinal analysis, we test the classifiers under the basic assumption, using the whole dataset: 70% for training and 30% for testing. We have to notice that the dataset will be imbalanced with respect to time (years). Despite the fact that we take 70% of all malware and 70% of all goodware, keeping the remaining 30% of the samples for testing purposes, the number of apps per year is not controlled, which can create an imbalanced effect. The aim of this scenario is to mimic the conditions that are commonly assumed in the literature.

Table 2 provides an overview of the detection results achieved by LongCGDroid with

different classifiers. For comparison, we also consider the three abstraction modes: P, C and M.

Table 2. Performance evaluation for models trained in different modes with the whole dataset.

Classifier	Abstraction	Accuracy (%)	Precision (%)	TPR (%)	FPR (%)	F1-score (%)
CNN	***P***	92.93	93.6	92.17	6.3	92.88
	C	96.01	95.63	96.43	4.4	96.03
	M	99.42	99.32	99.52	0.68	99.42
SVM	***P***	92.68	92.52	92.86	7.5	92.69
	C	94.98	95.01	94.95	4.98	94.98
	M	97.76	97.24	98.32	2.79	97.77
RF	***P***	93.09	93.07	93.12	6.93	93.09
	C	94.69	94.68	94.71	5.31	94.69
	M	97.99	97.45	98.56	2.57	98
LR	***P***	92.74	92.56	92.96	7.47	92.76
	C	91.41	91.23	91.63	8.8	91.43
	M	93.04	92.77	93.36	7.27	93.06
RNN	***P***	90.32	90.29	90.36	9.71	90.32
	C	90.26	90.25	90.28	9.75	90.26
	M	92.77	92.49	93.1	7.55	92.79
KNN	***P***	90.99	90.88	91.13	9.14	91
	C	90.16	90.11	90.23	9.9	90.17
	M	90.37	90.16	90.62	9.88	90.39

The analysis of the presented classifiers and their performance across various abstraction levels yields valuable insights into their effectiveness for Android malware detection. First, as we can see in Figure 10, the abstraction modes P, C and M are all effective for malware detection. Whatever the classifier, all the metrics: Accuracy (Figure 10(a)), Precision (Figure 10(b)), TPR (Figure 10(c)) and F1-score (Figure 10(d)) exceed 90%. Among the tested classifiers, CNN emerges as a robust contender, demonstrating consistently strong results across all three levels of feature abstraction. In particular, we note the CNN's ability to maintain both higher TPRs (Figures 10(c)) and F1-scores (Figures 10(d)), accentuating its accuracy in reducing false positives and maximizing true positives, while maintaining a strong balance between precision and TPR.

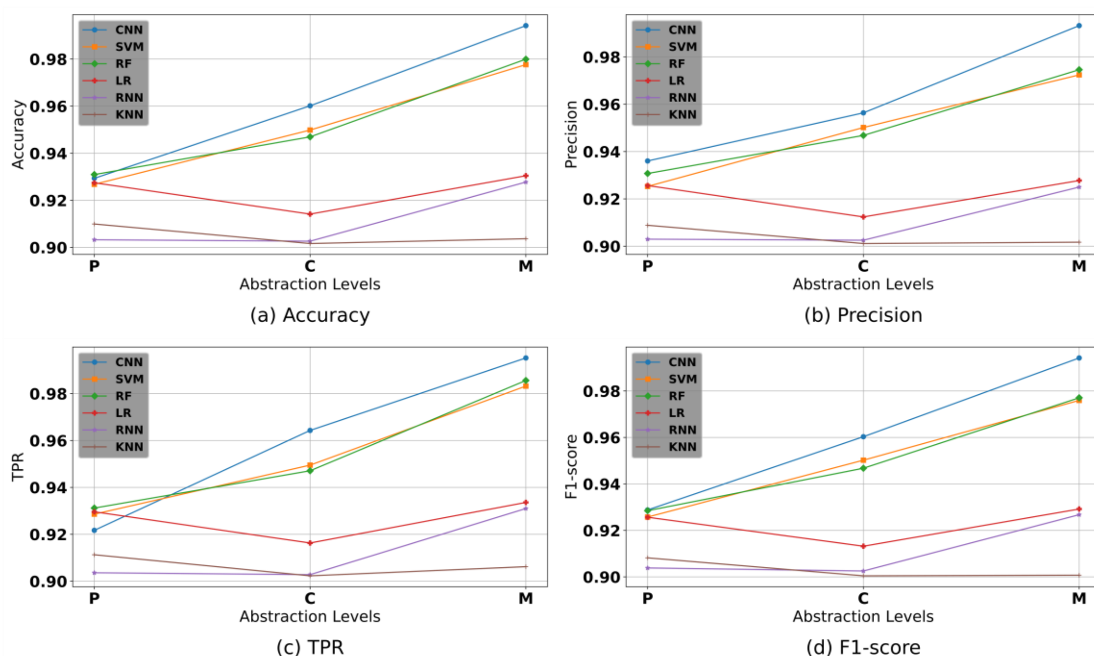

Figure 10. Results of the comparison of LongCGDroid with all classifiers and all abstractions.

This is most noticeable in the M abstraction, where the CNN has an F1-score of 99.42%. On the other

hand, the SVM and RF classifiers display performances close to those of the CNN. The SVM, for example, maintains consistent precision and TPR measurements, which indicates its reliability at different levels of abstraction. The RF classifier proves to be more sensitive to changes in abstraction, its F1-score varying from 93.09% with the P abstraction to 97.99% with the M abstraction, which indicates a dependence with regard to the level of abstraction chosen for the characteristics. The LR classifier shows less competitive results, with its F1-score almost constantly around 93%, which suggests its inability to find a balance between accuracy and TPR. Lastly, the RNN and KNN classifiers show the lowest results, whatever the level of abstraction used. Thus, CNN, SVM and RF all appear to be promising candidates, especially when high TPR and F1-score are important. However, no conclusions can be drawn about the robustness of the models.

5.2 Experiment 2: Longitudinal Performance of LongCGDroid

In this longitudinal study, all the classifiers are trained on the dataset from 2013 to 2018 and evaluated on the remaining part, from 2019 to 2022. The performance of various machine-learning and deep- learning classifiers was assessed across different levels of abstraction, as was the case with experiment 1.

Starting with the results from the first year of test 2019, it can be observed from Figure 11 that across all levels of abstraction, the CNN consistently achieved high TPR, Precision and F1-score values. Particularly, when considering the C abstraction, the CNN demonstrated remarkable performance with TPR and F1-score around 97%. The SVM showcased competitive results, although its performance slightly degraded with the increased abstraction. RF exhibited stable performance across different abstraction levels, while RNN and KNN demonstrated decreased results, with RNN showing relatively better performance in precision than KNN.

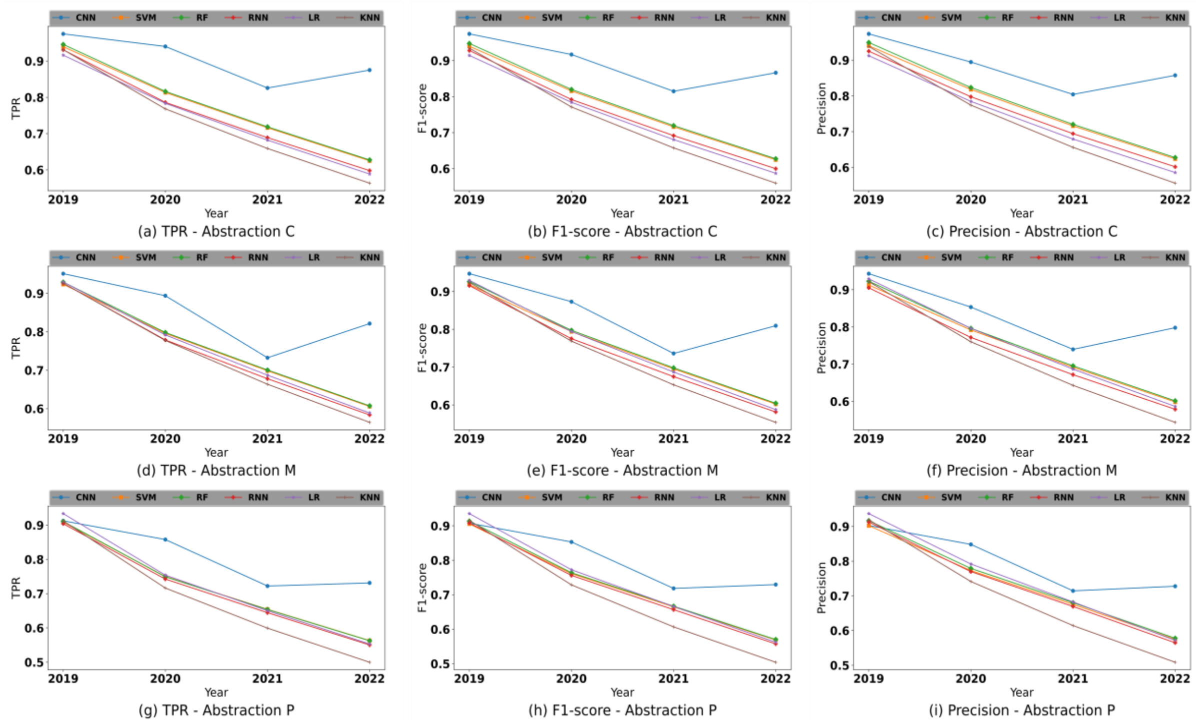

Figure 11. Results of the longitudinal comparison of LongCGDroid with all classifiers.

In the subsequent years, the trends shifted. By 2020, there is a general decrease in performance for most classifiers, especially in terms of accuracy and precision. However, CNN continued to outperform other classifiers in terms of TPR and F1-score, even though its precision has dropped. SVM, RF and RNN showcased consistent trends across the years, with SVM's performance being stable, but comparatively lower, RF maintaining relatively strong performance and RNN's accuracy and precision experiencing a slight decrease. This degradation is more important in package and method abstraction than in class abstraction.

Moving to 2021, there was a continued decline in overall performance. Notably, the decline was

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 09, No. 04, December 2023.

in the accuracy and precision of all classifiers. The performance gaps between classifiers became narrower and SVM's performance even improved in terms of precision with class and method abstractions. This year marked a more competitive landscape among the classifiers.

Finally, in 2022, the performance of CNN evolved once again. However, it shows signs of decline with P and M abstractions. SVM and RF exhibited a consistent trend with slight performance dips, while the performance of RNN and KNN showed more fluctuations.

The results show a consistent decline in model performance at all levels of abstraction. However, a notable observation is the gradual decline observed over time when using class abstraction. This particular abstraction shows a trend toward better results during the four-year post-training period. Class abstraction appears to be more resilient to API changes than the highest level of abstraction; namely, method abstraction. In contrast, package abstraction continues to show promising model performance. However, it is clear that these models lose stability over the years. Therefore, their effectiveness in representing the dataset is somewhat compromised.

5.3 Experiment 3: LongCGDroid *vs.* State-of-the-art Static Baseline Systems

To demonstrate the effectiveness of our proposed method in addressing malware evolution, we conducted a comparative study with eight state-of-the-art baseline systems; namely, Maldozer, DroidSieve, RevealDroid, MamaDroid, DroidEvolver, Admat, which are static approaches' classifiers, and DroidSpan and DroidCat, which are dynamic approaches' classifiers. We specifically choose to compare with approaches that (1) employ API calls to perform detection, (2) are fully designed considering the evolution of the apps and (3) are longitudinally evaluated. These criteria apply to Maldozer, DroidSieve, RevealDroid, MamaDroid, DroidEvolver, Admat, DroidSpan and DroidCat, except for DroidSieve and AdMat, which are not longitudinally evaluated in the original works. For consistency with the evaluated systems, we consider the configurations that produced optimal results in the respective original studies. For instance, it's important to highlight that MaMaDroid employs two distinct abstraction modes: family and package. The family mode operates at a more refined granularity level compared to the package mode, resulting in a smaller count of API calls, specifically nine families. On the other hand, Maldozer adopts only the abstraction method mode. Specifically, this involves employing the family abstraction combined with the RF model for MaMaDroid and utilizing the method abstraction for Maldozer. Regarding our approach, we will take into account the CNN model with the class abstraction. During this experiment, a subset of the dataset from 2013 to 2018 is used for feature extraction and training, while the remainder from 2019 to 2022 is used for evaluation. We focused our analysis on key performance indicators, specifically TPR and the F1-score.

The results indicate a notable consistency in the performance of LongCGDroid, especially when compared to other systems. One year after training, LongCGDroid outperformed all other systems with a TPR of 97.49% (Figure 12(a)) and a relatively low FPR of 2.66% (Figure 12(b)), achieving an F1-score of 97.42% (Figure 12(c)). MaMaDroid, DroidEvolver, DroidSpan and DroidCat also showcased commendable performance with F1-scores of 95.08%, 94.40%, 92.46% and 91.30%, respectively, which quantify their ability to correctly identify malware instances, while DroidSieve and RevealDroid lagged with an F1-score of 60.31% and 72.98%, respectively. AdMat, on the other hand, had a performance with an F1-score of 88.08%, placing it in the mid-range among the compared systems.

As we progress over the years, we observe a gradual performance decrease, as shown by the changes in the F1-score, with LongCGDroid still maintaining a leading position with an F1-score of 91.69% in 2020. Maldozer, DroidEvolver, MaMaDroid, DroidSpan and DroidCat followed closely with F1-scores of 88.57%, 87.56%, 86.94%, 82.73% and 85.67%, respectively. The performance gap between LongCGDroid and other systems like DroidSieve, RevealDroid and Admat widened, as they reached F1-scores of 41.67%, 66.49% and 77.54%, respectively. The trend of declining performance continued into 2021, with LongCGDroid recording an F1-score of 81.48% and all the other systems experiencing a decline to values below 80%. They appear to be significantly more impacted by the reduction in TPR. However, the performance variations among the systems were more apparent, with LongCGDroid maintaining its competitive edge. These results accentuate the resilience of the LongCGDroid. Figure 12(b) does not follow clear, decreasing trends,

as we can notice a little increase in the year 2022 on all the systems; this can be explained by the fact that at some point, an old behavior becomes popular again, leading to a sudden increase in the performance of all detectors.

We can see that DroidEvolver, MaMaDroid, DroidSpan and DroidCat remain more longitudinally competitive in comparison with our LongCGDroid system than the other systems. This can be attributed to the fact that MaMaDroid uses a lower granularity API family, DroidEvolver continually updates its feature set as an online learning system and both DroidSpan and DroidCat employ dynamic features to profile the apps, hence enhancing the characterization of their behavior, unlike Maldozer, which relies on a method-level abstraction. On the other hand, DroidSieve and AdMat, which were not initially designed to consider the evolution of the apps, displayed a consistent downward trend in their F1-scores, indicating diminishing effectiveness over the years. Although DroidSieve's and RevealDroid's initial study suggests that they are very resistant to obfuscation, their characteristics do not necessarily make them suitable choices for overcoming the evolution of apps. The decline and instability shown in the AdMat can be attributed to its exclusive reliance on the CFG inferred from the APIs without incorporating the DFG derived from pseudo-dynamic analysis, as is the case with LongCGDroid.

LongCGDroid is a static-based system that uses pseudo-dynamic analysis to extract Android API features, which provides an efficient mechanism to abstract significant characteristics of the Android applications under study, thus allowing us to extract Android API features efficiently while minimizing the overhead typically associated with real dynamic-analysis methods. However, using a real dynamic analysis could provide a more accurate reflection of the run-time behavior of the applications under analysis. The pseudo-dynamic analysis may not capture the real-time interactions and the exact execution flow in a similar manner as real dynamic analysis would. For instance Android malware may disguise its malicious behavior by only triggering a malicious API call after receiving a specific network signal. Under the current pseudo-dynamic analysis, such behavior might go unnoticed, as the analysis relies on a predetermined application state. However, with dynamic analysis, the system can observe the application's behavior in a controlled, yet realistic, execution environment, recording how it interacts with system and user data through API calls when stimulated by external triggers.

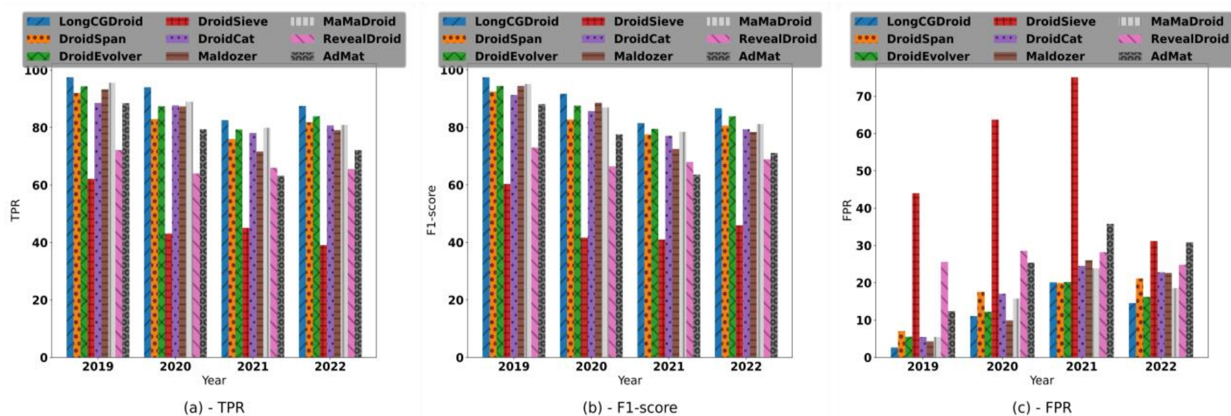

Figure 12. Performance evolution of the systems across the years.

6. CONCLUSION

In this work, we conducted a comparative longitudinal study between machine-learning and deep-learning algorithms using our adjacency matrix-based tool, LongCGDroid. Through our experiments using different levels of abstraction, we observed that the usual evaluation lacks information about model robustness. However, employing a longitudinal approach allows us to infer various details regarding model robustness and evolution. Thus, we observed that while the method abstraction yields better results in the usual approach, for maintaining higher robustness, it is advisable to focus on class abstraction. Class abstraction, being more resilient, enables balanced detection performance. This resilience has proven to be beneficial for the deep-learning CNN classifier for the sake of conserving a good balance of detection over time. Our method is also

compared with eight state-of-the-art baseline research works; our approach has proven to be more resilient over the years.

Future work may investigate the use of RGB color images to get more inferred details. This can have an impact on mitigating the diminishing performance of the classifiers of machine learning and deep learning-based malware detectors.

ACKNOWLEDGEMENTS

The authors would like to thank Salim Grabssi, the Director of the Computing Center of the University of Boumerdes, for letting them conduct some of the computing for this project.

REFERENCES

[1] Statista.com, "Mobile Operating Systems' Market Share Worldwide from January 2022 to January 2023," [Online], Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile- operating-systems-since-2009/.

[2] Security List by Kaspersky, "IT Threat Evolution Q1 2023, Mobile Statistics," [Online], Aailable: https://securelist.com/it-threat-evolution-q1-2023-mobile-statistics/109893/.

[3] W. Wang et al., "Constructing Features for Detecting Android Malicious Applications: Issues, Taxonomy and Directions," IEEE Access, vol. 7, pp. 67602-67631, 2019.

[4] Y. C. Shyong, T. H. Jeng and Y. M. Chen, "Combining Static Permissions and Dynamic Packet Analysis to Improve Android Malware Detection," Proc. of the 2nd Int. Conf. on Computer Communication and the Internet (ICCCI), pp. 75-81, Nagoya, Japan, 2020.

[5] L. Li, T. F. Bissyandé, M. Papadakis et al., "Static Analysis of Android Apps: A Systematic Literature Review," Information and Software Technology, vol. 88, pp. 67-95, 2017.

[6] H. Cai, "Embracing Mobile App Evolution *via* Continuous Ecosystem Mining and Characterization," Proc. of the IEEE/ACM 7th Int. Conf. on Mobile Software Engineering and Systems, pp. 31–35, Seoul, Korea, 2020.

[7] H. Cai and B. Ryder, "A Longitudinal Study of Application Structure and Behaviors in Android," IEEE Transactions on Software Engineering, vol. 47, no. 12, pp. 2934-2955, 2021.

[8] H. Cai, X. Fu and A. Hamou-Lhadj, "A Study of Run-time Behavioral Evolution of Benign *versus* Malicious Apps in Android," Information and Software Technology, vol. 122, p. 106291, 2020.

[9] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder and L. Cavallaro, "TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time," Proc. of the 28th USENIX Conf. on Security Symposium, pp. 729–746, 2019.

[10] L. Nguyen-Vu, J. Ahn and S. Jung, "Android Fragmentation in Malware Detection," Computers & Security, vol. 87, p. 101573, 2019.

[11] G. Suarez-Tangil and G. Stringhini, "Eight Years of Rider Measurement in the Android Malware Ecosystem: Evolution and Lessons Learned," IEEE Transactions on Dependable and Secure Computing, vol. 19, pp. 107-118, 2018.

[12] A. Guerra-Manzanares and H. Bahsi, "On the Relativity of Time: Implications and Challenges of Data Drift on Long-term Effective Android Malware Detection," Computers & Security, vol. 122, p. 102835, 2022.

[13] F. Ceschin et al., "Fast & Furious: On the Modeling of Malware Detection As an Evolving Data Stream," Expert Systems with Applications, vol. 212, p. 118590, 2023.

[14] L. Onwuzurike et al., "MAMADROID: Detecting Android Malware by Building Markov Chains of Behavioral Models," ACM Transactions on Privacy and Security, vol. 22, no. 2, pp. 1-34, 2019.

[15] E. B. Karbab, M. Debbabi, A. Derhab and D. Mouheb, "Android Malware Detection Using Deep Learning API Method Sequences," Digital Investigation, vol. 24, pp. S48-S59, 2017.

[16] K. Xu et al., "DroidEvolver: Self-Evolving Android Malware Detection System," Proc. of the 2019 IEEE European Symp. on Security and Privacy (EuroS&P), pp. 47-62, Stockholm, Sweden, 2019.

[17] G. Suarez-Tangil et al., "DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware," Proc. of the 7th ACM on Conference on Data and Application Security and Privacy, pp. 309–320, DOI: 10.1145/3029806.3029825, 2017.

[18] L. N. Vu and S. Jung, "AdMat: A CNN-on-Matrix Approach to Android Malware Detection and Classification," IEEE Access, vol. 9, pp. 39680-39694, 2021.

[19] J. Garcia, M. Hammad and S. Malek, "Lightweight, Obfuscation-resilient Detection and Family Identification of Android Malware," Proc. of the 40th Int. Conf. on Software Engineering, p. 497, DOI: 10.1145/3180155.3182551, 2018.

[20] H. Cai, N. Meng, B. Ryder and D. Yao, "DroidCat: Effective Android Malware Detection and Categorization *via* App-Level Profiling," IEEE Transactions on Information Forensics and Security, vol.

14, no. 6, pp. 1455-1470, 2019.
[21] H. Cai, "Assessing and Improving Malware Detection Sustainability through App Evolution Studies," ACM Trans. Softw. Eng. Methodol., vol. 29, no. 2, p. Article 8, 2020.
[22] N. Elenkov, Android Security Internals: An In-Depth Guide to Android's Security Architecture, ISBN-10: 9781593275815, No Starch Press, 2014.
[23] A. Desnos. "Androguard-reverse Engineering, Malware and Goodware Analysis of Android Applications," [Online], Aailable: https://github.com/androguard/androguard.
[24] S. Arzt et al., "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," SIGPLAN Not., vol. 49, no. 6, pp. 259–269, 2014.
[25] R. Nix and J. Zhang, "Classification of Android Apps and Malware Using Deep Neural Networks," Proc. of the Int. Joint Conf. on Neural Networks (IJCNN), pp. 1871-1878, Anchorage, USA, 2017.
[26] S. Y. Yerima and S. Khan, "Longitudinal Performance Analysis of Machine Learning-based Android Malware Detectors," Proc. of the 2019 Int. Conf. on Cyber Security and Protection of Digital Services (Cyber Security), pp. 1-8, Oxford, UK, 2019.
[27] J. Jung, H. Kim, D. Shin, M. Lee, H. Lee, S.-j. Cho and K. Suh, "Android Malware Detection Based on Useful API Calls and Machine Learning," Proc. of the 2018 IEEE 1st Int. Conf. on Artificial Intelligence and Knowledge Engineering (AIKE), pp. 175-178, Laguna Hills, USA, 2018.
[28] N. Peiravian and X. Zhu, "Machine Learning for Android Malware Detection Using Permission and API Calls," Proc. of the 2013 IEEE 25th Int. Conf. on Tools with Artificial Intelligence, pp. 300-305, Herndon, USA, 2013.
[29] M. Qiao, A. H. Sung and Q. Liu, "Merging Permission and API Features for Android Malware Detection," Proc. of the 2016 5th IIAI Int. Congress on Advanced Applied Informatics (IIAI-AAI), pp. 566-571, Kumamoto, Japan, 2016.
[30] A. H. E. Fiky, A. Elshenawy and M. A. Madkour, "Detection of Android Malware Using Machine Learning," Proc. of the IEEE Int. Mobile, Intelligent and Ubiquitous Computing Conf. (MIUCC), pp. 9- 16, Cairo, Egypt, 2021.
[31] M. Alazab et al., "Intelligent Mobile Malware Detection Using Permission Requests and API Calls," Future Generation Computer Systems, vol. 107, pp. 509-521, 2020.
[32] Z. Wang, K. Li, Y. Hu, A. Fukuda and W. Kong, "Multilevel Permission Extraction in Android Applications for Malware Detection," Proc. of the 2019 Int. Conf. on Computer, Information and Telecommunication Systems (CITS), pp. 1-5, Beijing, China, 2019.
[33] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an and H. Ye, "Significant Permission Identification for Machine Learning-based Android Malware Detection," IEEE Transactions on Industrial Informatics, vol. 14, no. 7, pp. 3216-3225, 2018.
[34] C. Yang, Z. Xu, G. Gu, V. Yegneswaran and P. Porras, "DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications," Proc. of the European Symposium on Research in Computer Security (ESORICS 2014), vol. 8712, pp. 163-182, 2014.
[35] T. E. Wei et al., "DroidExec: Root Exploit Malware Recognition against Wide Variability *via* Folding Redundant Function-relation Graph," Proc. of the 17th Int. Conf. on Advanced Communication Technology (ICACT), pp. 161-169, PyeongChang, Korea, 2015.
[36] A. Narayanan, L. Yang, L. Chen and L. Jinliang, "Adaptive and Scalable Android Malware Detection through Online Learning," Proc. of the Int. Joint Conf. on Neural Networks (IJCNN), pp. 2484-2491, Vancouver, Canada, 2016.
[37] Y. Wu, J. Shi, P. Wang, D. Zeng and C. Sun, "DeepCatra: Learning Flow- and Graph-based Behaviours for Android Malware Detection," IET Information Security, vol. 17, no. 1, pp. 118-130, 2023.
[38] T. Lei, Z. Qin, Z. Wang, Q. Li and D. Ye, "EveDroid: Event-aware Android Malware Detection against Model Degrading for IoT Devices," IEEE Internet of Things Journal, vol. 6, no. 4, pp. 6668-6680, 2019.
[39] J. McGiff et al., "Towards Multimodal Learning for Android Malware Detection," Proc. of the Int. Conf. on Computing, Networking and Communicat. (ICNC), pp. 432-436, Honolulu, USA, 2019.
[40] D. Li et al., "Opcode Sequence Analysis of Android Malware by a Convolutional Neural Network," Concurrency and Computation: Practice and Experience, vol. 32, no. 18, p. e5308, 2020.
[41] X. Sun et al., "Android Malware Detection Using Sequential Convolutional Neural Networks," Journal of Physics: Conference Series, vol. 1168, no. 6, p. 062010, 2019.
[42] N. McLaughlin et al., "Deep Android Malware Detection," Proc. of the Seventh ACM on Conf. on Data and Application Security and Privacy, pp. 301–308, DOI: 10.1145/3029806.3029823, 2017.
[43] T. H.-D. Huang and H.-Y. Kao, "R2-D2: ColoR-inspired Convolutional NeuRal Network (CNN)-based AndroiD Malware Detections," arXiv: 1705.04448 [cs.CR], 2018.
[44] P. Faruki, B. Buddhadev, B. Shah, A. Zemmari, V. Laxmi and M. S. Gaur, "DroidDivesDeep: Android Malware Classification *via* Low Level Monitorable Features with Deep Neural Networks," Proc. of the Int. Conf. on Security & Privacy (ISEA-ISAP 2019), vol. 939, pp. 125-139, 2019.
[45] K. Allix, T. F. Bissyandé, J. Klein and Y. L. Traon, "AndroZoo: Collecting Millions of Android Apps for the Research Community," Proc. of the 2016 IEEE/ACM 13th Working Conf. on Mining

Software Repositories (MSR), pp. 468-471, Austin, USA, 2016.
[46] VirusShare, "VirusShare.com - Because Sharing is Caring," [Online], Available: https://virusshare.com/.
[47] Virustotal, "Analyse Suspicious Files," [Online], Available: https://www.virustotal.com/.
[48] A. Salem, S. Banescu and A. Pretschner, "Don't Pick the Cherry: An Evaluation Methodology for Android Malware Detection Methods," arXiv: 1903.10560 [cs.CR], 2019.
[49] Android, "Android APIs Reference," [Online], Available: https://developer.android.com/reference/packages.

ملخص البحث:

تستهدف هذه الورقة البحثية مقارنة الأداء الطّولي بين مُصنِّفات تعلُّم الآلة ومُصنِّفات التّعلُّم العميق لكشْف البرمجيات الخبيثة في تطبيقات أَنْدرويد، باستخدام مستوياتٍ مختلفة من تجريد السِّمات. وباستخدام مجموعة بيانات لتطبيقات أَنْدرويد على مدى عشر سنوات (2013-2022).

نقترح طريقةً فعالةً قائمة على الصُّوَر لكشْف البرمجيات الخبيثة في تطبيقات أَنْدرويد. كذلك نعمل على تقييم الأداء الطّولي للطّريقة المقترحة، باستخدام نماذج تعلُّم الآلة ونماذج التّعلُّم العميق.

وقدّ بينت التّجارب تراجعاً متواصلاً في الأداء لجميع المصنِّفات عند تقييمها على عيّناتٍ من فتراتٍ متأخِّرة. واثبت نموذج التعلُّم العميق القائم على الشّبكات العصبية الالتفافية (CNN) تحت تجريد الصِّنْف قدْراً من الاستقرار مع الزّمن. وبمقارنة الطّريقة المقترحة بعددٍ من الطُّرُق المشابهة الواردة في أدبيات الموضوع، وأثبتت الطّريقة المقترحة تفوُّقها من حيث دِقّة الكشْف.

This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).