Android X-Ray - A System for Malware Detection in Android Apps using Dynamic Analysis

DAKSHINAMOORTHY KARTHIKEYAN, ARUN SIVAKUMAR,
CHAMUNDESWARI ARUMUGAM
Department of Computer Science and Engineering,
Sri Sivasubramaniya Nadar College of Engineering, Tamil Nadu,
INDIA

Abstract: - In recent years, mobile malware takes anywhere between several hours to several days to screen an app for malicious activity. More than 6000 apps are added to the Google Play Store everyday on average. Security analysts face an uphill battle against malware developers as the complexity of malware and code obfuscation techniques are constantly increasing. Currently, most research focuses on the development and application of machine learning techniques for malware detection. However, their success has been limited due to a lack of depth in the data sets available for training models. This paper uses a new method of Dynamic Analysis for Android apps to extract large amounts of information on the behavior of any app which can then be used for training models or to enable security analysts to take an informed decision quickly.

Key-Words: - Malware, dynamic analysis, security, android
Received: September 6, 2022. Revised: October 30, 2022. Accepted: November 6, 2022. Published: November 7, 2022.

1 Introduction

In this modern age of Software and fast moving technology, hackers are becoming more and more innovative in hiding the malicious activity of their software from users as well as traditional tools. Tools like antivirus software's are based on older techniques such as Hash comparison, crowdsourcing, system resource usage monitoring or network flow analysis. They may also make use of code analysis algorithms including system call identification, runtime library usage, or embedded string search. So the modern problem requires a modern solution.

Over 2.5 billion Android mobile active devices exist as per statistics record in 2019, [1]. Recent research, [2], has pointed out that close to 80% of Android malwares are hidden inside seemingly benign apps distributed through public app stores like the google play store. A report from McAfee Labs shows that 7.4 million components of mobile malware were identified in 2018, with an increase of 82% from last quarter of 2017, [3]. Today, the existing permissions-based app cautions the individuals about the consent required by an app before installing it, [4]. Therefore, the responsibility falls to the owners of app stores to ensure that such apps do not appear on their stores. This job is typically carried out by highly trained security analysts and the majority time is spent on analyzing the app to get information about its behaviors, like which files it accessed or network activity. By utilizing the dynamic analysis method in this work, this time can be cut down from several hours to a few minutes.

Many researchers have provided Android malware security solutions using dynamic analysis method, [5]. Dynamic analysis method examines the behavior of malware at runtime, [6], and makes it challenge-able to adversary, [7]. This method can control malware when executed using a sandboxed environment or emulator, [8]. It discovers malware when the application is in execution state and acquires the data pertaining to the application's behavior by setting up an environment using sandbox, virtual machine and other forms, [9].

The main objective of this proposed work is to provide a tool where the user can upload any Android app and select a list of malware's to scan for, or suspicious app behaviors to check for. The app will be installed and executed on a virtual machine running the modified version of Android Operating System(OS). The user will be able to interact with the virtual machine, while the background activity of the app is monitored. If any of the security rules are triggered or the app is seen accessing unauthorized resources, the tool will flag the app as a malware and inform the user.

The major contributions in this paper are as follows.

- Users can upload an android app and scan a list of existing malware's for detecting suspicious behaviors.

- The executable file, android app is analyzed to monitor the behavior and actions of the app to detect the existing malware.
- Interact and monitor the background activity of the app. Any violation of the system will flag the app as a malware.
- Further on detection, an alert is notified to the user about the existing malware.

The organization of the work proceeds as follows. Section 2 introduces the literature survey, which contains the new technology, research, and methodology used in the existing work associated with this work and Section 3 discusses the system design, system components, resources and components. Section 4 discusses the experimental setup and workflow. Section 5 summaries the results and discussion on the analysis of the malware. Finally, with future work the paper is concluded.

2 Literature Survey

Singh et. al., [10], described the taxonomy of features that may be used by a machine learning approach to dynamic analysis of malware: Memory and registers usage, instruction traces, network traffic and API call traces. T. Cho et. al., [11] performed a security based assessment with various code obfuscation techniques. While there has been some research into the field, [12], [13], it is made difficult by the fact that each program uses a different variant of the obfuscation scheme with varying levels of code 3 complexity. Mercaldo et. al., [14], conducted an analysis of a dataset containing 20,000 Android applications, both malware and benign on the basis of the APIs used.

Even small changes in the code like hiding the API call behind an additional layer of obfuscation is sufficient to fool an algorithm-based anti-malware solution, the proof of which is seen in semantics of internet security threat report, [15], showing that 246 million variants of existing malware's were seen in 2018 alone. The hybrid static approach introduced by Kim et. al.,[16], would go a long way in improving the scalability of the project. Regarding the previous research into automating the malware detection process, Viet Duc et. al.,[17], introduced a neural network to identify malwares that made use of 7 types of features.

All research to date using Application Programming Interface(API) calls as a feature in classification appears to use only the names of the methods or class in which they are defined. This approach may produce a good result in the research context but it may not be practically applicable to the thousands of apps that need to be scanned every day. The reason for this is that there are millions of functions within Android OS used by benign and malicious apps. Malicious behaviour does not come from the mere calling of the function, but it arises from how, or for what purpose the method is being called. Such data is typically impossible to retrieve in any other OS, but thanks to the open source nature of Android, it is possible to make modifications such that whenever an API is called, it will print out the values of the parameters to a serial console using Android's built-in logging system, at which point it can be extracted to create the report or train a model.

3 System Design

The system design, system components used and connection between them for detection is discussed here. Dynamic malware analysis is the process to detect malware by setting up an environment, identifying the malware behaviour patterns, [18], and analyzing the sequence of system calls invoked by malware, [19]. Chun et al., [20], designed an application layer software to integrate existing dynamic analysis frameworks.

Here, in this proposed system, a dynamic malware analysis approach is applied to detect malware and it is shown in Figure. 1. The user can upload any number of Android Package Kit(APK) files for testing in the frontend user interface. An Android virtual machine with versions of Android 10.0 and Linux Kernel 4.4 is used for execution and monitoring of the app. Many logs are collected during the installation process of the app. To identify the malware activity a Virtual Private Network(VPN) server is used as it takes a copy of network activity from the virtual machine and transfers the data to the log collection module for analysis. The analysis module classifies the collected data and reports to users in the frontend user interface.

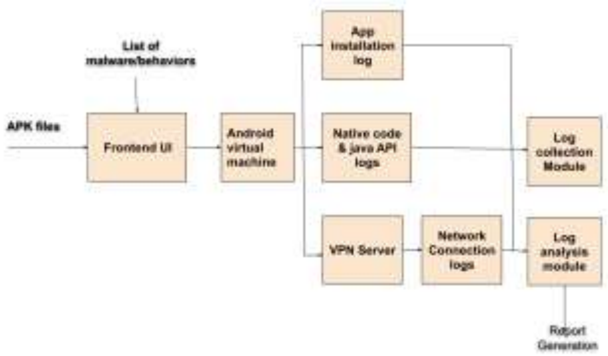


Fig. 1: Proposed system

Resources used in this work for detection of malware are as follows.

- Files opened/read/ deleted/modified
- Triggered a phone call / SMS
- Read call logs/sms
- Downloaded and installed another app
- Network access - list of URLs and files downloaded
- Hardware sensors accessed
- Notification access
- Permissions used.

The components used here for this work are represented in Figure 1. The brief description of the components is discussed below.

Frontend User Interface comprises the main homepage, where the user can upload any number of APK files for testing. Logging statements and condition checking blocks will be built into the implementations of all major APIs in Android and the Java Virtual Machine(JVM), as well as the linux kernel. The capability in the 'Native code and Java API Logs' component was built by patching various APIs in the JVM. The patches made to JVM are listed below:

- Redirected standard output and standard error streams to a separate stream monitored by the Collection module
- Patched the APIs in java.io package to trigger a message in X-Ray when any file is accessed.

Native code and Java API Logs provide access to all sensitive device information, such as the file system, radios, device sensors, communication with other apps, etc. To identify apps during execution activity, the backend will provide a VPN server as the only possible method of outside network access.

Listing 1.1:Input code

```
class helloWorld {
   public static void main(String[] args) {
      System.out.println("Hello");
   }
   public void testing(int x) {}
   private int privFun() {
      return -1;
   }
}
```

The backend VPN server forwards a copy of all network activity through network connection logs, from the virtual machine and stores it so that the data can be used for analysis. The log collection module collects and combines log data obtained from various sources like LogCat, liblog, the VPN Server and the kernel dmesg output into a single datastore for analysis and report generation. The analysis module is responsible for segregating records into categories like device sensors, file system access, inter-process communication, network access, etc. and classifying the records as malicious or safe activity. From the consolidated datastore of all log records, a comment is presented to the user.

Listing 1.2 Output code

```
import android.util.Log;
class helloWorld {
   public static void main(String[] args) {
      Log.xray("helloWorld", "main" + "|args=" + args);
      System.out.println("Hello");
   }
   public void testing(int x) {
    Log.xray("helloWorld", "testing" + "|x=" + x);
   }
   private int privFun() {
      return -1;
   }
}
```

4 Detection and Performance Attainment

The procedure for experimental setup to launch the sandboxed Android Virtual Machine is divided into five stages as follows.

- Download Android OS source code
- Set the build target to x86 Android Cuttlefish VM
- Add X-Ray logging code to OS
- Build the full OS and generate artifacts
- Launch the Cuttlefish VM with Virtual Network Computing(VNC) server.

After the download of the X-Ray modified Android OS source code, [21], this research work uses Android Cuttlefish Virtual Machine, [22], as the build target, as this enables multi-tenancy on the server and allows multiple VMs to run efficiently on the same server in parallel. These Android framework classes files are passed to the JavaParser module for generating Abstract Syntax Tree (AST).

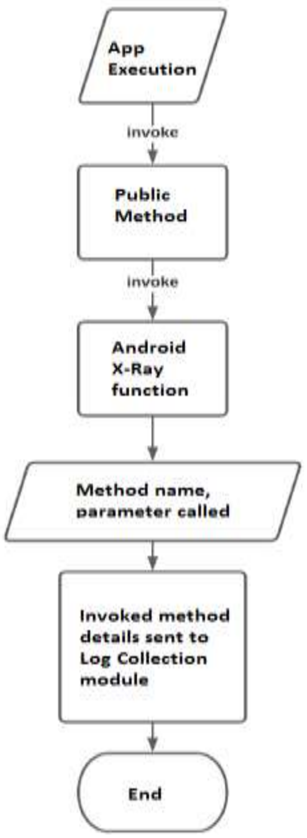

App Execution
invoke
Public Method
invoke
Android X-Ray function
Method name, parameter called
Invoked method details sent to Log Collection module
End

Fig. 2: Flowchart for addition of log statements in Java files

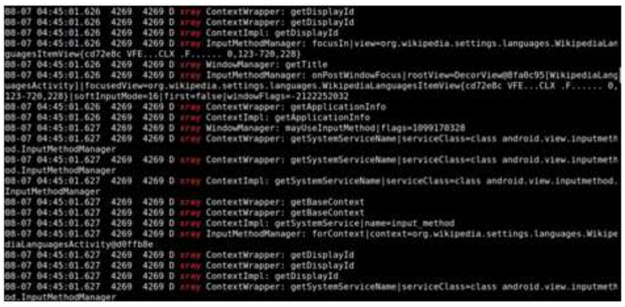
To build code generating software, each node of AST is traversed. If the node of AST contains a Public Method the MethodModifier function will prepend the Android X-Ray function as the first child of that node. By adding X-Ray function as the first child of any function, it can be ensured that the function will never be executed without also triggering the execution of the X-Ray function. The Android X-Ray function is a custom function that integrates with Android's logging system and will send the method name, parameters passed, parameter values to the back end via the log collection module.

The steps followed for modification of Android OS code is depicted as a flowchart in Figure. 2 and the sample java program and modification is listed in Listing 1.1 and Listing 1.2 respectively. Android uses a combination of GNUMake and the Google Soong, [23], to build a system for compiling the source code and generating system images to run on the virtual machine. The build system produces the following artifacts after compiling the code which are required for launching the Cuttlefish virtual machine. The Cuttlefish engine is launched with VNC server enabled that uses Kernel-based Virtual Machine(KVM), [24], emulation and launches as many instances as required by the system. It also sets up TAP network adapters, [25], that provide an internet connection to the machine by tunnelling the network traffic of each virtual machine independently through a separate adapter.

The APK file of the app to be tested is uploaded through the front-end web-page. App installation module uses Android Asset Packaging Tool (AAPT), [26], tool to extract app metadata including the package name and the list of activities. Each application is executed using the Monkey tool that generates pseudo-random streams of user events for about one minute generating 2000 gestures with a delay of 500ms between each event and the logs are collected. The log collection module spawns a subprocess to monitor the output of adb logCat. The module iterates through the log entries and removes any garbage logs which are not related to the X-Ray analysis system. It then uses the syntax of the ”Log.xray()” command to identify the API that triggered execution of each log, and applies a command-specific regular expression to those log entries corresponding to functions that have parameters with important values.

The log entries are then grouped by the source. The total number of occurrences of each unique class or function call is calculated and added as an attribute of the group. A benign app will typically have fewer frequency of API calls or permission requests than a malicious one. Most of the features required for malware classification algorithms are derived from the X-Ray output in a straightforward way. The vanilla Android OS comes with 4 million unique files loaded on the first boot, [27]. Therefore, from each log entry for access to a file, the features of root partition, file extension and file access in read or write mode were created. It is now possible to perform Categorical Encoding, [28], on the newly created features and the model has successfully represented the most important data from the file system access logs.

For the network activity logs, the X-Ray system monitors the console for any Domain Name Server (DNS) queries and saves the query hostname. It queries the IPVoid APIs to get up-to-date historical information on malicious activity from that address, along with details about how the address is being used. By summarizing the collected logs and performing feature extraction on the submitted APKs, it is possible to collect the data for each submitted app and prepare a dataset for training. Further, a random classifier, [29], then can be applied to classify a new app as malicious or benign based on the historical data.



```
08-07 04:45:01.626  4269  4269 D xray ContextWrapper: getDisplayId
08-07 04:45:01.626  4269  4269 D xray ContextWrapper: getDisplayId
08-07 04:45:01.626  4269  4269 D xray ContextImpl: getDisplayId
08-07 04:45:01.626  4269  4269 D xray InputMethodManager: focusIn|view=org.wikipedia.settings.languages.WikipediaLan
guagesItemView{cd72e8c VFE...CLX .F...... 0,123-720,228}
08-07 04:45:01.626  4269  4269 D xray WindowManager: getTitle
08-07 04:45:01.626  4269  4269 D xray InputMethodManager: onPostWindowFocus|rootView=DecorView@8fa0c95[WikipediaLang
uagesActivity]|focusedView=org.wikipedia.settings.languages.WikipediaLanguagesItemView{cd72e8c VFE...CLX .F...... 0,
123-720,228}|softInputMode=16|first=false|windowFlags=-2122252032
08-07 04:45:01.626  4269  4269 D xray ContextWrapper: getApplicationInfo
08-07 04:45:01.626  4269  4269 D xray ContextImpl: getApplicationInfo
08-07 04:45:01.627  4269  4269 D xray WindowManager: mayUseInputMethod|flags=1099170328
08-07 04:45:01.627  4269  4269 D xray ContextWrapper: getSystemServiceName|serviceClass=class android.view.inputmeth
od.InputMethodManager
08-07 04:45:01.627  4269  4269 D xray ContextWrapper: getSystemServiceName|serviceClass=class android.view.inputmeth
od.InputMethodManager
08-07 04:45:01.627  4269  4269 D xray ContextImpl: getSystemServiceName|serviceClass=class android.view.inputmethod.
InputMethodManager
08-07 04:45:01.627  4269  4269 D xray ContextWrapper: getBaseContext
08-07 04:45:01.627  4269  4269 D xray ContextWrapper: getBaseContext
08-07 04:45:01.627  4269  4269 D xray ContextImpl: getSystemService|name=input_method
08-07 04:45:01.627  4269  4269 D xray InputMethodManager: forContext|context=org.wikipedia.settings.languages.Wikipe
diaLanguagesActivity@d0ffb8e
08-07 04:45:01.627  4269  4269 D xray ContextWrapper: getDisplayId
08-07 04:45:01.627  4269  4269 D xray ContextWrapper: getDisplayId
08-07 04:45:01.627  4269  4269 D xray ContextImpl: getDisplayId
08-07 04:45:01.627  4269  4269 D xray ContextWrapper: getSystemServiceName|serviceClass=class android.view.inputmeth
od.InputMethodManager
```

Fig. 3: Raw console output of logging functions

5 Results and Discussion

The dataset is collected from 200 benign apps downloaded from Google Play Store and 150 malicious apps from the AndroZoo repository of malicious android applications, [30]. The AndroZoo dataset is a static dataset and is easily obtainable by anyone who wants to reproduce this experiment. Each app is run through all stages of the X-Ray system from app installation to log summarization, and the final data is saved into the dataset. In order to collect the log information at various points, certain changes need to be made to the Android OS so that the relevant data can be extracted during execution of the test app.

Summary of Resources Accessed by App under test

[illegible]

Fig. 4: Summary of output logs

The system takes advantage of Android's built in logging subsystem, [31], which can write any string data to standard output. The standard output is monitored by the log collection submodule. This module is responsible for identifying the source of each log entry, parsing the relevant data and sending the information to the analysis submodule for further processing. For making these changes to the source code, the system takes advantage of the open source library JavaParser, [32], which enables the system to programmatically parse, modify and save a java source code file.

Hence, the procedure for collecting log data is fairly simplified - a virtual machine running the modified OS is launched, and the test app is launched inside it while the standard output is monitored using Android's available LogCat tool, [33]. However, a log entry is created only when a function is actually executed, i.e. presence of a function call in an app does not guarantee that a corresponding log entry will be found as the app may execute a code path that doesn't contain the function call. To minimize this risk, the system makes use of another tool provided by Android known as Monkey, [34], a pseudo-random event generator that can randomly send touch and keyboard events to the app, increasing the coverage of the system. By implementing the various modules of the system as described in Figure. 1, it is possible to automate the collection and installation of apps submitted for testing, launch them in an Android VM and collect the output logs which are then summarized and displayed to the user as an alert message.

Sample outputs of the system on providing the wikipedia android app, [35], for analysis are described in this section. Firstly, a subset of the raw text data printed to standard output by the logging functions added to the OS. Figure 3 displays the raw console output of logging functions. This raw text is the input to the log collection module. Since each log entry follows a similar structure - the keyword "xray" followed by the class name and function name, followed by the parameter values separated with a '|' character, this module can easily identify the source of each message, extract the relevant data from the string and store it in a data structure more suited for analysis, namely JSON.

To further simplify the downstream analysis work, the JSON entries may be segregated into different categories at this stage, such as permission requests, DNS lookups, file access, etc. A sample of the JSON output derived at this stage is shown in Listing 1.3. A sample network activity summary is listed in Figure. 4. The extraction of features from Network Activity and File System Logs is represented in Figure. 5.

Dimensionality reduction and PCA transformation was applied on the features extracted from the X-Ray log dataset before training the model. PCA transformed X-Ray logs dataset is represented in Figure 6. Random forest classification model was used to classify the app malicious or benign. The F1 score obtained here in this process is 94.2%. Some challenges faced in this work are as follows.

- Parsing the very detailed logs output by Native code and Java API Logs module.
- Removing duplicate log entries (Android APIs may internally call other APIs, and

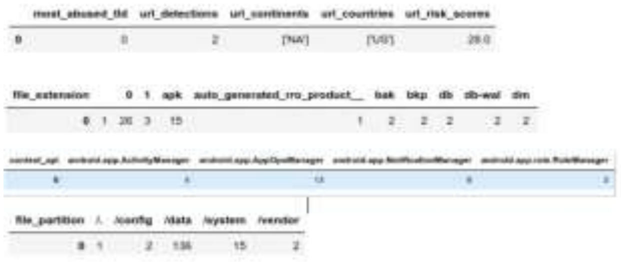
each call will output a unique X-Ray log message).

Listing 1.3 Log Collection output

```
{
 "opened_files": {
      "/data": 6,
      "/data/app": 1,
      "/data/app/vmdl1152042043.tmp": 5,
      "/data/system/install_sessions/app_icon.1152042043.png":
            4,  "/data/app/vmdl1152042043.tmp/base.apk": 9
             },
   "checked_permissions":{
"android.permission.ACCESS_NETWORK_STATE|app=org.wi
kipedia.beta":                                           10,
"android.permission.NETWORK_SETTINGS|app=org.wikipedi
a.beta": 2,
"android.permission.MAINLINE_NETWORK_STACK|app=org
.wikipedia.beta": 2,
"android.permission.INTERACT_ACROSS_USERS|app=org.wi
kipedia.beta": 1,
"android.permission.INTERACT_ACROSS_USERS_FULL|app
=org.wikipedia.beta":                                     1,
"android.permission.INTERNAL_SYSTEM_WINDOW|app=org
.wikipedia.beta": 1
   },
   "dns_lookups": [
      "meta.wikimedia.org",
      "en.wikipedia.org",
      "in.appcenter.ms",
      "upload.wikimedia.org"
   ]
}
```

Some of the identified shortcomings of the work are listed below.

1. **Log Collection:** This work provides a method of logging the execution of any Java API, however a large part of the OS including the linux kernel, drivers and low level APIs are written in C/C++. An approach similar to the proposed one using JavaParser may be employed on these classes to improve the scope of Log Collection.
2. **Noise Reduction:** Since a log event is triggered every time a function executes regardless of how the call was triggered, the generated logs tend to contain a high amount of noise. A mechanism to either avoid logging of calls known to be of normal behaviour, or to filter them out in a post-processing stage would improve the quality of data collected.
3. **AI Automation:** With the aggregation of log data for a large number of apps, an ML model may be trained to classify apps as malicious or benign using the dataset. It would then be possible to reduce the backlog of apps to be manually tested by filtering out the obviously malignant apps automatically.



	most_abused_tld	url_detections	url_continents	url_countries	url_risk_scores
0	0	2	['NA']	['US']	28.0

file_extension		0	1	apk	auto_generated_rro_product__	bak	bkp	db	db-wal	dm
0	1	20	3	15	1	2	2	2	2	2

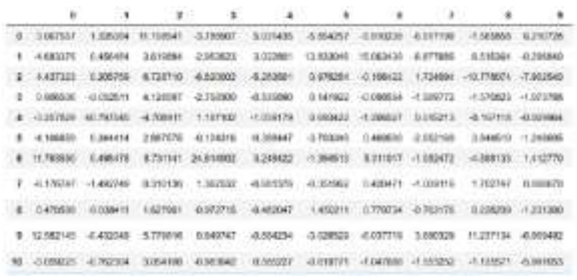
context_api	android.app.ActivityManager	android.app.AppOpsManager	android.app.NotificationManager	android.app.role.RoleManager
0	4	13	6	3

file_partition	/	/config	/data	/system	/vendor
0	1	2	136	15	2

Fig. 5: Extracted features from Network Activity and File System Logs

Few challenges left for future work are as follows.

- Distinguishing between logs output by an app
- Logs produced during normal system operation (Some progress was made using a Process ID (PID) filter, but there is still more work to be done).



	0	1	2	3	4	5	6	7	8	9
0	3.067537	1.335004	11.159541	-3.785907	5.031435	-5.554257	-0.590239	-6.517196	-1.585865	6.210726
1	-4.683375	0.456484	3.615894	-2.953623	3.032881	13.833046	15.063430	6.677885	8.518364	-0.295840
2	4.437323	0.305759	6.728710	-6.820002	-5.263881	0.978284	-0.198422	1.724894	-10.778074	-7.902540
3	0.686536	-0.052511	4.120087	-2.750900	-0.335060	0.141922	-0.098554	-1.389772	-1.370823	-1.573786
4	-3.357626	40.793540	-4.708911	1.187932	-1.034179	0.060422	-1.286027	0.016213	-8.157115	-0.029964
5	-4.166830	0.344414	2.667076	4.134316	-0.359447	-3.703340	0.466630	2.052198	3.544610	-1.246866
6	11.765590	0.496478	8.731141	24.814902	3.249422	-1.394613	5.211017	-1.032472	-4.389135	1.412770
7	-4.176747	-1.482749	0.310136	1.382532	-6.583576	-0.353562	0.408471	-1.009115	1.702747	0.008670
8	0.479530	0.038411	1.627061	0.972715	-8.462047	1.450211	0.770734	-0.763175	0.228299	-1.231380
9	12.582145	-6.432048	5.779816	0.049747	-0.504234	-3.028520	-6.037719	3.690329	11.237134	-6.009482
10	-3.059225	-4.762304	3.054186	-0.383842	0.565227	-0.018771	-1.047880	-1.555252	-1.133571	-5.991653

Fig. 6: PCA Transformed X-Ray logs dataset

6 Conclusion and Future Work

This work attempted to bridge the gap between the innovative tools available to hackers in preventing the detection of malware in Android apps and the corresponding tools available to the security analyst charged with defending the Android ecosystem from them. Security analysts needed to manually decompile and inspect the bytecode of every submitted app to extract the information of used APIs, accessed network and local system resources, requested permissions, etc. Whereas the malware developer can use any simple method such as obfuscation and redeploy the app without making any changes to the code. By using dynamic analysis combined with the X-Ray logging system and ADB monkey to speed up manual user interaction with the apps, the proposed system has been efficient in cutting down the time required for dataset preparation from several hours to a few minutes. While the system can significantly improve the efficiency and accuracy of malware detection in Android, there is a high scope for future works to build upon it.

References:

[1] Russell Brandom. (2019). The Verge Press article: There are now 2.5 billion active Android devices. https://www.theverge.com/2019/5/7/18528297/google-io-2019-androiddevices-play-store-total-number-statistic-keynote

[2] Kotzias Platon, Caballero Juan, and Bilge, Leyla. (2020). How Did That Get In My Phone? Unwanted App Distribution on Android Devices.

[3] McAfee Mobile Threat Report (2019). \\https://www.mcafee.com/enterprise/enus/assets/reports/rp-mobile-threat-report-2019.pdf

[4] Android (2022). Overview of Android Permissions Architecture and user approval process.https://developer.android.com/guide/topics/permissions/overview.

[5] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec*, April, 2013.

[6] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. *Proceedings of the Network and Distributed System Security Symposium* (NDSS), 2014.

[7] Ankita Kapratwar , “Static and Dynamic Analysis for Android Malware Detection”, San Jose State University, May 2016

[8] V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*. vol. 9, no. 1, pp. 99–108, Jan 2014.

[9] Mahima Choudhary, Brij Kishore, HAAMD:Hybrid Analysis for Android Malware Detection. *International Conference on Computer Communication and Informatics* (ICCCI -2018). 04 – 06, 2018, Coimbatore, INDIA.

[10] Jagsir Singh, Jaswinder Singh,(2020). A survey on machine learning-based malware detection in executable files, *Journal of Systems Architecture*. 101861, ISSN 1383-7621. https://doi.org/10.1016/j.sysarc.2020.101861.

[11] T. Cho, H. Kim and J. H. Yi. (2017). Security Assessment of Code Obfuscation Based on Dynamic Monitoring in Android Things. *IEEE Access*. vol. 5, pp. 6361-6371. doi: 10.1109/ACCESS.2017.2693388.

[12] Yadegari B., Johannesmeyer, B., Whitely B. and S. Debray. (2015). A Generic Approach to Automatic Deobfuscation of Executable Code. *IEEE Symposium on Security and Privacy*, San Jose, CA, pp. 674-691. doi: 10.1109/SP.2015.47.

[13] Z. Kan, H. Wang, L. Wu, Y. Guo and G. Xu. (2019). Deobfuscating Android Native Binary Code. IEEE/ACM 41st *International Conference on Software Engineering: Companion Proceedings* (ICSE-Companion), Montreal, QC, Canada, pp. 322-323, doi:10.1109/ICSE-Companion.2019.00135.

[14] Mercaldo Francesco, Di Sorbo Andrea, Visaggio Corrado Aaron, Cimitile, Aniello and Martinelli Fabio. (2018). An exploratory study on the evolution of Android malware quality. *Journal of Software: Evolution and Process*. 30. e1978.10.1002/smr.1978.

[15] Symantec.(2019) Symantec Internet Security Threat Report. https://docs.broadcom.com/doc/istr-24-2019-en

[16] D. Kim, D. Mirsky, A. Majlesi-Kupaei and R. Barua. A Hybrid Static Tool to Increase the Usability and Scalability of Dynamic Detection of Malware. *13th International Conference on Malicious and Unwanted Software* (MALWARE), Nantucket, MA, USA, 2018, pp. 115-123. doi: 10.1109/MALWARE.2018.8659373.

[17] Nguyen Viet Duc and Pham Thanh Giang. (2018). NADM: Neural Network for Android Detection Malware. *In The Ninth International Symposium on Information and Communication Technology* (SoICT 2018). Danang City, Viet Nam. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3287921.3287977

[18] T. Bell.(1999). The concept of dynamic analysis. *ACM SIGSOFT Softw. Eng. Notes*. vol. 24, no. 6, pp. 216–234, 1999, doi: 10.1145/318774.318944.

[19] Ori Or-Meir, Aviad Cohen, Yuval Elovici, Lior Rokach, Nir Nissim. (2021). Pay Attention: Improving Classification of PE Malware Using Attention Mechanisms Based on System Call Analysis, *IEEE International Joint Conference on Neural Networks* (IJCNN).

[20] Chun-Yi Wang, Chi-Yu You, Fu-Hau Hsu, Chia-Hao Lee, Che-Hao Liu, YungYu Zhuang.(2021). SMS Observer: A dynamic mechanism to analyze the behavior of SMS-based malware, *Journal of Parallel and Distributed Computing*. 156, 25–37.

[21] Repo - The Multiple Git Repository Tool. 2022. https://gerrit.googlesource.com/git-repo/

[22] Cuttlefish Virtual Android Devices - a configurable virtual Android device that can run both remotely. 2022. https://source.android.com/setup/create/cuttlefish-cts

[23] The Android Open Source Project. Soong Build System for flexible and fast code compilation.2022. https://source.android.com/setup/build

[24] Liu Di, Zhang Yun, Zhang, Ni and Hu, Kun. (2014). A Research on KVM Based Virtualization Security. *Applied Mechanics and Materials.* 543-547.3126-3129. 10.4028/www.scientific.net/AMM.543-547.3126.

[25] Ganguly Arijit Wolinsky, David Boykin, P. and Figueiredo, Renato. (2007). Decentralized Dynamic Host Configuration in Wide-Area Overlays of Virtual Workstations. 1-8. 10.1109/IPDPS.2007.370664

[26] Syaifudin, Yan. (2021). How does Android Testing Tool work? Case study: Robolectric. 0.13140/RG.2.2.19597.87523.

[27] The Android Open Source Project. Manifest of git repositories on Android OS https://android.googlesource.com/platform/manifest/

[28] Kedar Potdar, Taher Pardawala, and Chinmay Pai. (2017). A Comparative Study of Categorical Variable Encoding Techniques for Neural Network Classifiers. *International Journal of Computer Applications*. 175. 7-9.10.5120/ijca2017915495.

[29] Ho, Tin Kam (1995). Random Decision Forests (PDF). *Proceedings of the 3rd International Conference on Document Analysis and Recognition*, Montreal, QC, 14–16 August 1995. pp. 278–282. Archived from the original (PDF) on 17 April 2016. Retrieved 5 June 2016.

[30] Kevin Allix, Tegawend´e F. Bissyand´e, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: collecting millions of Android apps for the research community, *In Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. Association for Computing Machinery, New York, NY, USA, 468–471. DOI:https://doi.org/10.1145/2901739.2903508

[31] Kotecha Jay, and P, Prabu. (2018). An Investigation on android background services for controlling the unauthorized accesses using android LOG system. *International Journal of Engineering and Technology*. 7. 301. 10.14419/ijet.v7i2.6.11268.

[32] Hosseini, Roya and Brusilovsky, Peter. (2013). JavaParser: A Fine-Grain Concept Indexing Tool for Java Problems. *CEUR Workshop Proceedings*. 1009. 60-63.

[33] Jang, Hae-Sook. (2012). Android Log Cat Systems Research for Privacy. *Journal of the Korea Society of Computer and Information*. 17. 10.9708/jksci/2012.17.11.101.

[34] Hasan Hayyan, Ladani Behrouz and Zamani Bahman. (2020). Enhancing Monkey to trigger malicious payloads in Android malware. pp.65-72. 10.1109/ISCISC51277.2020.9261909.

[35] Wikipedia Android app (2022) - https://github.com/wikimedia/apps-android-wikipedia

Contribution of Individual Authors to the Creation of a Scientific Article (Ghostwriting Policy)

Dakshinamoorthy Karthikeyan, Arun Sivakumar implemented the entire project and drafted the research paper. Chamundeswari Arumugam was responsible for guiding, integrating and shaping this as a research paper.