Conception of an Autonomous Dynamic Analysis System for Android Malwares

Amel Boudrega, Selma Benzouaoua, Philippe Ea, Osman Salem, and Ahmed Mehaoua
Centre Borelli UMR 9010
Université Paris Cité
Paris, France
{firstname.lastname}@u-paris.fr

***Abstract*—This paper focuses on dynamic analysis for malware detection on Android. Initially, a literature review was conducted to understand both static and dynamic analysis approaches and their limitations, particularly highlighting the shortcomings of static analysis. The study demonstrates techniques for extracting various traces, such as system calls and network traffic, using dynamic analysis. The core of the study is the design of an automated system for the dynamic analysis of Android malware. This system automates the capture and analysis of APK traces using modules that monitor system calls, debug logs, and network traffic. It was found that relying on a single dynamic analysis module is insufficient, leading to false negatives, whereas combining data from all three modules enhances detection accuracy. Future directions include developing an intermediary using MQTT to reduce database load and improving the learning process for the three modules.**

***Index Terms*—Dynamic Analysis, Malware Detection, Android Security, Network Traffic Analysis, Machine Learning.**

I. Introduction

Smartphones have become indispensable tools in modern daily life, serving purposes ranging from entertainment and work to planning and note-taking. Central to their functionality is the Operating System (OS) they operate on, with Android standing out as a prominent open-source OS in this domain. Developed and maintained by Google, Android has been widely adopted across the mobile platform industry, powering approximately 69.74% of smartphones worldwide. This widespread adoption fosters a unified user experience, minimizing discrepancies across devices from different manufacturers and ensuring consistent access to application libraries and graphical interfaces.

However, Android's popularity has also made it a prime target for malware developers seeking to exploit the vast amounts of personal data stored on these devices. According to the CIA model (Confidentiality, Integrity, Availability), malware is defined as any code aimed at compromising:

- Confidentiality, i.e., the protection of sensitive data on the system.
- Integrity, i.e., the origin of the data and the level of protection of these data.
- Availability, i.e., the ability to use a system when desired.

Traditional approaches to malware detection are rooted in two primary methodologies: Static Analysis (SA) and Dynamic Analysis (DA). These methodologies are essential for comprehensively assessing the security of software systems, particularly in the context of Android malware.

SA constitutes the examination of an application's components without the need for execution. This method delves into the structural elements of the application, such as its binary code, software libraries, configuration files, and resources like images and string data. By examining these components, SA aims to uncover potential vulnerabilities or malicious code patterns that could compromise the confidentiality, integrity, or availability of the device and its data.

In contrast, DA involves the observation and evaluation of an application's behavior during its execution. This real-time assessment captures and analyzes runtime activities such as network communications, system calls, and interactions with external entities. DA is particularly effective in detecting malicious behavior that may manifest only when the application is running, such as unauthorized data exfiltration, interaction with suspicious servers, or anomalous system modifications. By actively monitoring the application's runtime environment, DA provides a deeper understanding of its operational behavior and enables the detection of sophisticated malware that could evade SA scrutiny.

Most approaches can be classified wholly or partly into one of these categories. The methodologies differ in the type of elements used for the analysis, the processing done on these elements to determine the presence of malicious behavior, and their resilience to evasion attempts commonly found in Android malware.

This paper aims to contribute to the field by proposing a specialized system for Android malware analysis. Specifically designed for analysts, this system focuses on DA of network traffic, system calls, and debug logs to identify and isolate suspicious behavior. The structure of this paper is as follows: Section II reviews existing literature in Android malware analysis, Section III outlines our dataset and methodology, Section IV presents our findings and analysis, and Section V concludes with a discussion on future research directions.

II. Related Work

A. Static Analysis

SA is the examination of the entirety of an application's content to infer its behaviors without having to observe it in execution. This type of approach avoids executing the malware

on a real or simulated platform, thus limiting the risk of infection and not requiring specific hardware resources for the task (e.g., an Android phone). On Android, this approach typically involves analyzing the content of the application's APK. This section details the main techniques for acquiring the artifacts required for SA, as well as the proposed methods to analyze them and identify malicious behaviors.

In the context of SA on Android, the application's source code (Java and native C/C++), Dalvik/compiled instructions, or an intermediate representation of the application's code are all potential candidates for analysis.

Using an application's Java source code allows for the use of generic techniques targeting Java in general, which can be generalized for the Android platform. An example of this approach is the Tai-e framework by Tan *et al.*, in [1], which allows for optimization, instrumentation, analysis, and visualization of Java applications. Its features that operate on Java source code have been adapted to be compatible with Android programs. The source code of an application can offer indications of the developer's intent through comments and the structure of the code itself. However, this approach is limited if the source code is not available or has been obfuscated by a tool. The assets (or resources) contained in the application can also be extracted for analysis.

In [2], Elsersy *et al.* demonstrated that this approach is commonly applied in antivirus software. Their study shows that these tools use assets to recognize files typically contained in known malware (e.g., images, text files, etc.).

The application manifest is also a target of interest for static analysis. Several studies have used the permissions listed in the manifest to determine if an application has a permission pattern similar to that of one or more malware [3], [4]. Other studies, such as [5] by Liu *et al.*, have used the declaration of components in the manifest to determine if an application publicly exposes components that could potentially leak sensitive data.

Several SA techniques on Android have been proposed. Notably, marker analysis and permission analysis have contributed to the improvement of statically detected malware on Android.

B. Limitations of static analysis

The use of SA cannot guarantee that all possible behaviors of an application during execution can be statically reached. On Android, this issue is especially prevalent since Java allows dynamic code loading. Android is a system where the expression of certain behaviors relies on the occurrence of events external to the applications (e.g., a user pressing a help button). Some of these behaviors only occur in the presence of actions that can be asynchronous and unpredictable. This event-driven model can dynamically affect the application's execution flow. For example, receiving an SMS from a command and control server could contain an encryption key used by the application during its execution to dynamically decrypt and execute malicious content. In this scenario, SA would not be able to analyze the encrypted content if the encryption key is strong enough, as it is external to the malware code. This type of scenario is a real constraint of SA, necessitating another approach to address it.

C. Dynamic Analysis

DA is a detection technique aimed at evaluating malware by executing the application in a real environment. The main advantage of this technique is that it detects dynamic code loading and records the application's behavior during execution.

The diagnostic features of the operating system involve using all available tools within the OS to extract the dynamic content necessary for analysis. These can include utilities and methods provided by the Linux kernel. The complete list of tools is accessible via the command "ls/system/bin" on an Android device. The "ps" utility can obtain the hierarchical list of all processes and threads if the "-t" option is included. "lsof" can retrieve all open files (virtual or physical) by the running processes on the system, and the process ID controlling them is also listed. "netstat" lists TCP or UDP sockets used or previously used for communication, their status, and the owning process. "strace" captures all system calls made by an application.

Several studies have utilized the observation of outgoing and incoming network communications of the device in the context of malware analysis on Android [6]–[8]. Acquiring the data transmitted over the network can be done by listening to it from an agent external to the Android device, capturing all traffic with a network trace capture and analysis tool like Wireshark.

System call interception targets the methods of the Linux kernel. Intercepting system calls provides visibility into all processes and operations of the OS. This technique is subject to performance constraints, as the additional processing load associated with logging can slow down the system.

Several DA techniques on Android have been proposed. Notably, system call analysis and the analysis of the application's inputs/outputs have contributed to the improvement of dynamic malware detection on Android.

System call analysis infers an application's behavior by observing the sequence of system calls, the content passed as arguments, and their return values. The work of Obaidat *et al.* [9] and Su *et al.* [10] used system call analysis to identify sequences representing behaviors deemed of interest, such as those used in sending a text message. Similarly, Alazab *et al.* [11] exploited this type of analysis to extract the frequency of different types of system calls for applications known to be benign, based on their category. From these results, they were able to establish that malware does not exhibit the same pattern (types of calls and their frequency) as a benign application belonging to the same category.

Given the effectiveness of DA compared to SA, we propose an automated system for DA of Android malware, the design of which will be detailed in the following section.

III. Proposed Approach

Our approach is a three-phase analysis, as illustrated in Figure 1. We conducted a DA using various tools to record their different impacts on an Android emulator. The results of this phase will be stored in files, and subsequently, the most relevant information will be extracted from these traces.

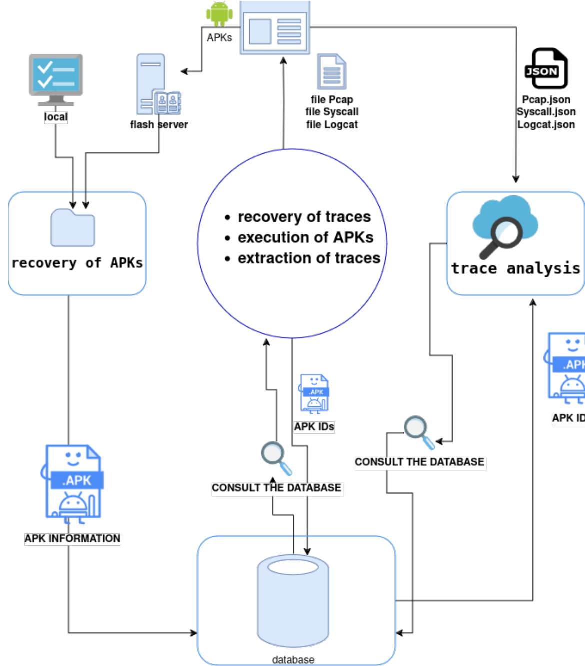

Fig. 1: The Architecture of our System

A. Retrieving the APKs

First, we have to retrieve the APKs. This phase will automatically perform the following steps for each APK loaded locally or by the user on our website hosted on a Flask web server as shown in Figure 2.

- Retrieve its MD5 and SHA1 hashes.
- File Hashes: The hash value, also known as the file fingerprint or checksum of the application, is obtained using cryptographic hash functions such as MD5, SHA1, and SHA256.
- Calculate its ID by incrementing the value in the "FolderID" file.
- Retrieve the Timestamp.
- Insert this information into the "Traces" table in our database specified for the APKs to be analyzed.
- Create a folder named with the APK ID in the folder dedicated to traces.

B. Trace Retrieval

To accommodate many applications, our trace retrieval phase implements two automatic processes: an application execution process and, in parallel, a trace extraction process.

The Android SDK provides an AVD manager, which allows the creation, modification, and deletion of Android emulators.

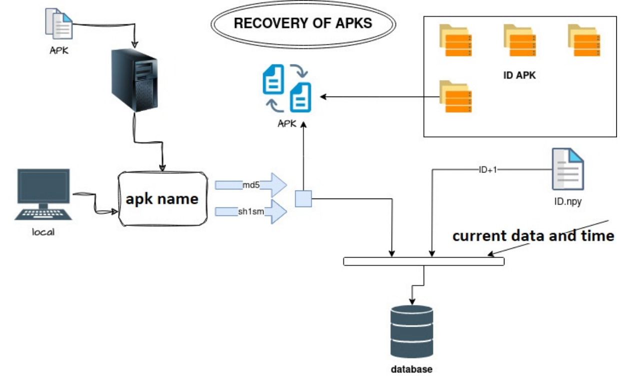

Fig. 2: Process to Retrieve the APKs

Various properties can be set for an AVD, such as the Android version, screen resolution, and available sensors. Before starting the different processes, we configured our Android emulator as shown in Table I. This configuration allows us to design and revert our emulator to its initial state after each analysis. The trace retrieval phase begins by creating a clean "snapshot" of our virtual machine. By default, this snapshot contains only what is installed by default in a new Android emulator created from Android Studio, allowing for the quick launch of our emulator. This phase uses adb to enable communication between the host machine and the emulator. From there, we start the application and immediately begin tracing various behaviors using multiple tools, such as logcat, MonkeyRunner, strace, and tcpdump.

TABLE I: Configurations Used by our Emulator

Platform	Android Studio 2.1
Device	Pixel 5
Target	Android 9.0 x86 API Level 28
RAM	1536 MB
SD Card	521 MB

The adb tool is a command-line utility that communicates with Android devices, allowing for tasks such as listing devices, managing applications, and transferring files by running as a client-server program with components on both the development machine and the Android device.

MonkeyRunner is a tool that runs on Android devices to generate a specified number of pseudo-random user and system events for stress-testing applications, with customizable options for event types and crash handling.

Android does not have a runtime console to display traces of the progress status. Instead, it offers a "logcat" event log that allows viewing messages from various applications and the system. This log provides a means to debug the application. It is part of the adb tools and allows us to:

- View, filter, and collect all application and system traces.
- Retrieve all unexpected errors generated during the application's execution.

Strace is a diagnostic, instructional, and debugging tool useful for system administrators and troubleshooters to trace

system calls and signals in programs, aiding in bug isolation, integrity verification, and capturing race conditions by examining the user/kernel interface.

Android tcpdump is a command-line packet capture utility. It can capture packets from Wi-Fi connections, cellular connections, and any other network connection on an Android device.

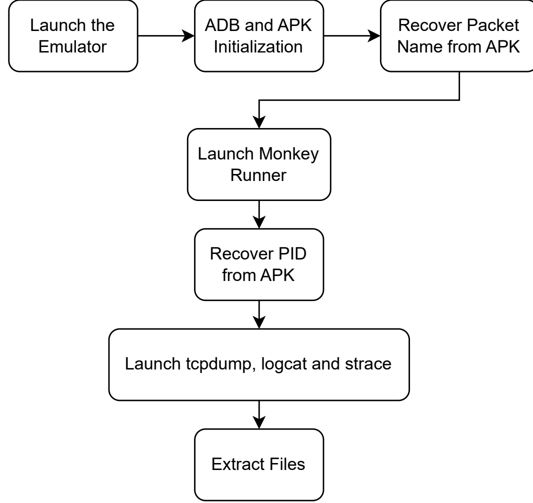

Fig. 3: Trace Extraction

To collect traces from an APK, we perform the following steps:

- Retrieve the list of unanalyzed APK IDs from the database.
- Interact with ADB and the connected emulator.
- Execution of the detailed steps of the "Trace Extraction" process is illustrated in Figure 3.

When the collection duration ends, the files are extracted from the SD card of the emulator to the folder corresponding to the APK trace. Once trace collection is complete, the database is updated by adding the collection duration/date and noting whether the analysis was successful. If any issues occur, a text file is created detailing the anomaly.

C. Extraction of Analyses

As explained previously, this phase uses the files generated by the previous phase, namely "logcat.txt", "pcap.pcap", and "syscall.txt", which are present in each successfully traced APK folder. After each trace is completed, this step automatically extracts precise information from each file that we deem relevant. This phase consists of three analyzers for the different files: debug journal file, network traffic, and system calls.

1) Analysis of Logcat: The "logcat.txt" file contains a large amount of logs, where each line represents a log message. Each log message includes the date and time, PID-TID, priority, and a tag. The tag of a system log message is a short string indicating the system component from which the message originates (e.g., ActivityManager). After careful consideration, we decided to filter these messages, focusing on permissions and URLs.

Permissions are used to protect the privacy of a user on an Android device. Android applications must request permission to access sensitive user data, such as contacts and SMS messages. Depending on the functionality, the system may automatically grant permission or prompt the user to approve the request. It's important to note that no application is allowed to perform operations that would negatively impact other applications, the operating system, or the user. Examples of potentially dangerous permissions include reading or writing private user data such as contacts or emails, reading or writing files of another application, creating network access, keeping the device awake, etc. (see Table II).

TABLE II: Permissions Judged as Dangerous

Permission Group	Permissions
CALENDAR	READ CALENDAR WRITE CALENDAR
CALL LOG	READ CALL LOG WRITE CALL LOG PROCESS OUTGOING CALLS
CAMERA	CAMERA
CONTACTS	READ CONTACTS WRITE CONTACTS GET ACCOUNTS
LOCATION	ACCESS FINE LOCATION ACCESS COARSE LOCATION
MICROPHONE	RECORD AUDIO
PHONE	READ PHONE STATE READ PHONE NUMBERS CALL PHONE ANSWER PHONE CALLS ADD VOICEMAIL
SMS	SEND SMS RECEIVE SMS READ SMS RECEIVE WAP PUSH RECEIVE MMS
STORAGE	READ EXTERNAL STORAGE WRITE EXTERNAL STORAGE

URL stands for Uniform Resource Locator. A URL is simply the address of a given resource, unique on the Web. Malicious URLs pose a very common and serious threat to security. They can be used to lure users into becoming victims when visiting phishing sites, downloading spam, and other content, which can result in privacy breaches, financial loss, or the installation of malware on the user's device. In our case, URLs are used to communicate with the developer/attacker of the malware. That's why we decided to retrieve every URL the application attempted to connect to, with filtering based on its PID.

2) Analysis of Syscalls: Android is an open-source mobile telephony platform based on Linux. The system call is the interface provided by the Linux kernel. The Linux operating system has approximately 300 system calls, which can be classified based on their functionalities. Thus, there are system calls responsible for critical activities such as process management, memory, and device handling. For example, a fake installer malware can affect the user with financial loss by sending a premium SMS. The highly invoked system calls during SMS sending include:

- "sendto()", "recefrom()", which are used for sending and receiving data from the socket.
- Process control-related system call such as "ptrace()" is used for tracing and controlling other processes, and "sigprocmask()" is used for process signal blocking, "wait4()", "futex()", "getpid()" to obtain the process ID, "getuid()" to get the user ID of the process owner, "prctl()" to control process execution.
- System calls related to reading and writing data from files stored on the phone and SD memory such as "write()", "read()", "ioctl()", "fcntl64()", "stat64()", "close()", "open()", "mmap()", "munmap()", "lseek()", "dup()", etc.

Malicious and benign applications exhibit different behaviors. For example, malicious Android applications request more permissions or access sensitive resources more frequently. Therefore, after capturing the system calls involved in the application, we calculated the frequency of each system call invoked by the malicious code process and recorded the result in a JSON file. Of course, the precise sequence of generated system calls will vary depending on the random selection made by the MonkeyRunner. However, the frequency of different system calls is relatively stable for a given application. The frequency representation of system calls carries information about the application's behavior. A particular system call may be more heavily used in a malicious application than in a benign one, and the frequency representation of system calls is intended to capture this information.

3) Network Trafic Analysis: We have also proposed an approach based on network traffic, assuming that malicious traffic differs from normal traffic. We designed this analysis to examine captured data from traffic generated by malware samples in a real internet environment for a duration of 10 minutes. In TCP/UDP traffic, there are several pieces of information that can be extracted. To do this, we calculated the average packet size and grouped packets into sessions. For each TCP/UDP session, we extracted the following fields: Source IPs, Destination IPs, Source Ports, Destination Ports, Protocol Identifiers, Cumulative TCP Flags, Layer 7 Protocols, Incoming Bytes, Outgoing Bytes, Incoming Packets, Outgoing Packets, Flow Duration. The Domain Name System (DNS) is one of the core protocol suites of the Internet, responsible for directing requests for Internet resources to the absolute hosting machine, and these resources are identified by URLs that include domain names, some of which may be malicious. For DNS traffic, we retrieved the following information: domain name, message type (query or response), and its type as described in Table III.

Type	Description
A	Enables mapping a DNS name to an IPv4 address
AAA	Enables mapping a DNS name to an IPv6 address
CNAME	Enables mapping between two DNS names

TABLE III: Examples of DNS Record Types

IV. Experimental Results

We assume that the presence of a malicious domain name in network traffic and malicious URLs, along with dangerous permissions in logcat, is sufficient to consider the application malicious. Figure 4 presents a histogram depicting the frequency of various system calls for four different applications. The histogram reveals distinct patterns in system call usage between legitimate applications (Duolingo and WhatsApp) and malware (Gamecenter and hua.ru.quan). Malware applications tend to show extreme values for certain system calls ("read" in the case of Gamecenter and "write" for hua.ru.quan), which could be leveraged as indicators for detecting malicious activities. These insights into system call behaviors can inform the development of DA tools for malware detection, highlighting specific system call sequences and frequencies as potential markers for identifying malicious software.

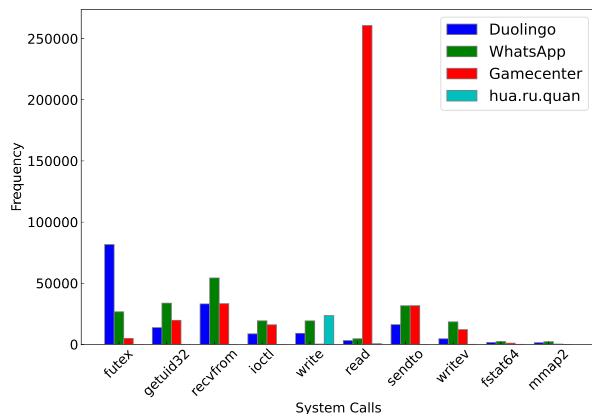

Fig. 4: Frequencies of System Calls for Two Legitimate Software and Two Malware

Given this ambiguity, we concluded that the use of Machine Learning (ML) is indispensable for this stage of analysis. We proposed a learning model that combines the Synthetic Minority Over-sampling Technique (SMOTE) with a ML classifier to detect malicious behaviors in Android applications.

The dataset consisted of legitimate and malicious samples obtained from our system. We labeled legitimate samples as class 0 and malicious samples as class 1. Missing columns in the dataset were handled by replacing them with mean values.

We also applied the SMOTE to handle class imbalance in the training data. This will help us to avoid potential issues such as overfitting. Random Forest (RF), Support Vector Machine (SVM), Logistic Regression (LR), and Gradient Boost (GB) models were trained on the resampled training data.

To evaluate the trained models' performances, we used various metrics, including accuracy, precision, recall, F1-score, ROC curve, and Area Under the Curve (AUC).

- Accuracy measures the overall correctness:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

where TP (True Positive) is the number of correctly classified positives, TN (True Negative) is the number of

correctly classified negatives, FP (False Positive) is the number of negatives incorrectly classified as positives, and FN (False Negative) is the number of positives incorrectly classified as negatives.

- Precision measures the proportion of true positives out of predicted positives:

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

- Recall measures the proportion of true positives out of actual positives:

$$Recall = \frac{TP}{TP + FN} \tag{3}$$

- F1-score is the harmonic mean of precision and recall:

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{4}$$

We conducted our experimentation on the resampled dataset we mentioned earlier, with 20% of the data used for validation and 80% for training. Figure 5 illustrates the ROC curves for four different models. Table IV summarizes the performance metrics for each model.

TABLE IV: Detailed Results (%) for our Models

Model	Accuracy	Precision	Recall	F1-score	Time (s)
RF	88.70	87.88	89.20	88.55	0.14
LR	58.21	42.86	91.17	63.20	0.01
SVM	77.50	73.33	76.67	74.44	0.34
GB	72.59	72.59	71.27	71.63	0.26

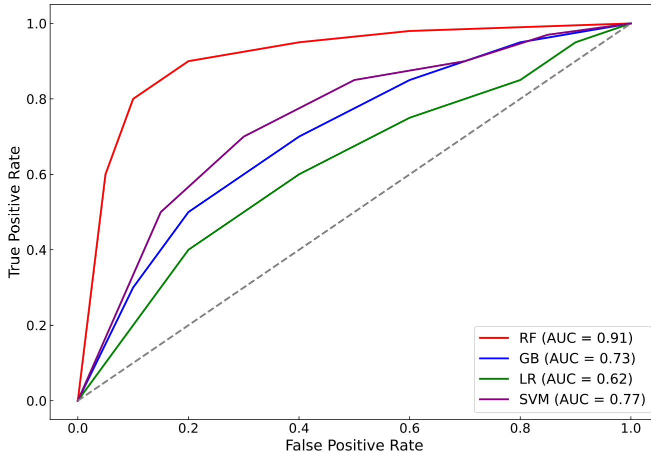

Fig. 5: ROC Curve for our Models

The RF model demonstrates the best overall performance across multiple metrics. This indicates that RF is the most effective model for this classification task, offering a high level of accuracy and reliability in predictions. Although LR has the shortest training time, its lower performance metrics suggest it is less suitable for this specific application. SVM and GB offer balanced performance but do not surpass RF.

These results highlight the importance of evaluating multiple metrics to select the most appropriate model for a given task. The RF model's strong performance across all considered metrics makes it the best choice for this analysis.

V. Conclusion

This study demonstrated an approach to detecting malicious behaviors in Android applications using DA and ML across three modules: syscall, logcat, and network traffic. Despite comprehensive system construction and promising initial results, the ML model focusing on syscall data alone did not achieve a high level of distinction between benign and malicious behaviors. This emphasizes the need for a more integrated approach combining all three modules for effective malware detection.

As a future perspective, we intend to expand our ML framework to encompass the analysis of logcat and network traffic data. By integrating these additional modules, we aim to create a comprehensive and multi-faceted detection system that leverages the strengths of DA across different data sources. This holistic approach is expected to significantly enhance the robustness and accuracy of malware detection systems, proving the importance of our initial idea to analyze and apply ML to syscall, logcat, and pcap data collectively. This will not only improve detection accuracy but also provide deeper insights into the behavioral patterns of malicious applications, thereby contributing to more effective cybersecurity measures.

References

[1] T. Tan and Y. Li, "Tai-e: A developer-friendly static analysis framework for java by harnessing the good designs of classics," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1093–1105.

[2] W. F. Elsersy, A. Feizollah, and N. B. Anuar, "The rise of obfuscated android malware and impacts on detection methods," *PeerJ Computer Science*, vol. 8, p. e907, 2022.

[3] Y. Sharma and A. Arora, "A comprehensive review on permissions-based android malware detection," *International Journal of Information Security*, pp. 1–36, 2024.

[4] D. Ö. Şahin, O. E. Kural, S. Akleylek, and E. Kılıç, "A novel permission-based android malware detection system using feature selection based on linear regression," *Neural Computing and Applications*, pp. 1–16, 2023.

[5] Z. Liu, L. F. Zhang, and Y. Tang, "Enhancing malware detection for android apps: Detecting fine-granularity malicious components," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1212–1224.

[6] W. Wang, D. Tian, W. Meng, X. Jia, R. Zhao, and R. Ma, "Msym: A multichannel communication system for android devices," *Computer Networks*, vol. 168, p. 107024, 2020.

[7] 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.

[8] A. Muzaffar, H. R. Hassen, M. A. Lones, and H. Zantout, "An in-depth review of machine learning based android malware detection," *Computers & Security*, vol. 121, p. 102833, 2022.

[9] I. Obaidat, M. Sridhar, K. M. Pham, and P. H. Phung, "Jadeite: A novel image-behavior-based approach for java malware detection using deep learning," *Computers & Security*, vol. 113, p. 102547, 2022.

[10] X. Su, L. Xiao, W. Li, X. Liu, K.-C. Li, and W. Liang, "Droidportrait: android malware portrait construction based on multidimensional behavior analysis," *Applied Sciences*, vol. 10, no. 11, p. 3978, 2020.

[11] M. Alazab, M. Alazab, A. Shalaginov, A. Mesleh, and A. Awajan, "Intelligent mobile malware detection using permission requests and api calls," *Future Generation Computer Systems*, vol. 107, pp. 509–521, 2020.