IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 2, July 2026
Impact Factor: 8.2

APK Sentinel: An Automated Framework for Static and Dynamic Security Scanning and Report Generation of Android APK Files

Amulya L D[1], Akshatha R[2], Prof. Mahalakshmi M[3]
[*1,2]Students, Department of Computer Application
[*2]Assistant Professor, Department of Computer Application
Maharaja Institute of Technology, Mysore, Karnataka, India

Abstract: *Android applications commonly request permissions and expose internal components without users having any practical way to judge whether those requests are reasonable or risky. This paper presents APK Sentinel, a web-based system that performs both static and dynamic analysis of an uploaded Android APK file and converts the technical findings into a report that a non-expert can understand. During static analysis, the APK is decompiled with Apktool and its AndroidManifest.xml is examined for requested permissions and exported activities, services, broadcast receivers and content providers. During dynamic analysis, the same application is installed and run inside an isolated Android emulator so that network activity, file access and permission usage can be observed under controlled conditions. Findings from both phases are combined into a single weighted risk score, categorised as Low, Medium or High risk, and an integrated large-language-model assistant is used to suggest plain-language remediation steps for each flagged issue. The system was implemented using Python, Flask, React.js and a SQLite/MySQL backend and was validated against a mixture of benign and known-malicious APK samples, with results cross-checked against MobSF for consistency*

Keywords: *Android security, APK analysis, static analysis, dynamic analysis, permission classification, malware detection*

I. INTRODUCTION

Whenever an Android user installs an application, the system usually asks for a list of permissions, and almost nobody reads through that list carefully before tapping install. The application itself may work perfectly well, but underneath it may be requesting far more access than it needs, or leaving one of its own internal components exposed to other apps on the device. Both situations are common causes of data leakage and malware infection on Android, and neither is obvious to an ordinary user simply by looking at the app or its description on a store listing.

Security researchers already have tools that perform static and dynamic analysis of APK files, MobSF being one of the more widely used examples, but these tools are generally built with a security professional as the intended audience. Their output is technically accurate but dense: permission names, manifest flags, component identifiers and CVSS-style scores that mean very little to a student, a hobbyist developer, or a person who simply wants to know if an app is safe to install. This gap between what security tools can detect and what an average user can actually interpret is the problem this project set out to address.

Prior work in this space has approached the problem from different angles. Enck et al. [1] examined the foundations of the Android permission model, while Zhou and Jiang [2] characterised how malware families evolved and what behavioural patterns they shared. Arp et al. [3] proposed DREBIN, which used static features extracted from the manifest and disassembled code for lightweight malware detection, and Felt et al. [4] studied how well permission warnings were actually understood by real users, concluding that most were not. Kabakus et al. [6] and later Mohamad Arif et al. [7]

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 2, July 2026
Impact Factor: 8.2

built on the same idea of permission-centric analysis, showing that even a fairly simple, rule-based inspection of an app's declared permissions can already flag a meaningful share of risky applications without needing a full dynamic run. What most of this literature has in common is a focus on detection accuracy rather than on how the result is explained back to the person who has to act on it, which is the gap APK Sentinel tries to fill.

The objective of this project was therefore to design a system that performs a reasonably thorough static and dynamic assessment of an uploaded APK, but presents its conclusions in ordinary language, with a risk label a person can act on and, where possible, a suggestion of what to actually do about the issue found. The intended audience is students, junior developers and everyday users rather than seasoned security analysts, so the emphasis throughout the design has been on making the report readable without stripping out the underlying technical rigour.

II. METHODOLOGY

The system follows a two-stage analysis pipeline. An uploaded APK first goes through static analysis, which inspects the application without executing it, and is then, where the workflow calls for it, installed and run inside a sandboxed emulator for dynamic analysis. The results of both stages feed into a single scoring module, which produces one consolidated report rather than two separate outputs the user has to reconcile themselves.

A. Static Analysis. Static analysis begins by decompiling the uploaded APK with Apktool, which reconstructs the AndroidManifest.xml and the application's resource files without needing to run any of the app's actual code. The manifest is then parsed to extract every permission the app declares, and each permission is classified according to Android's own protection levels, normal, dangerous or signature, so that permissions with a genuinely higher potential for misuse (contacts, SMS, location, storage) are weighted more heavily than routine ones (internet access, vibration). In parallel, the manifest is scanned for activities, services, broadcast receivers and content providers that have been marked as exported, since a component left exposed by mistake can often be invoked by another app on the same device without the user's knowledge.

B. Dynamic Analysis. Where static analysis alone leaves a question mark, for example an app that requests a permission but whose actual use of that permission cannot be determined from the manifest, the same APK is installed on an isolated Android emulator and run for a fixed observation window. Android Debug Bridge (ADB) is used to install the app, launch it and pull logs while it executes, and the runtime session is monitored for network requests, file-system access, permission usage that actually occurs (as opposed to permission that is merely declared) and any behaviour that looks anomalous relative to what the app claims to do. Because this entire phase runs inside the emulator rather than on a host machine, even a genuinely malicious sample cannot affect the system performing the analysis.

C. Risk Scoring and AI-Assisted Remediation. Findings from the static and dynamic stages are combined by a scoring module that assigns a numeric weight to each issue, dangerous permissions and exposed components contribute more than benign findings, and sums these weights into a single overall score. This score is then mapped onto a three-level label (Low, Medium or High risk) so that the headline result of a scan is something a user can act on immediately, before reading any of the supporting detail. Each individual finding is also passed, along with a short prompt, to a large-language-model API, which returns a plain-language explanation of the risk and a concrete suggestion of what the user or developer might do about it, for instance recommending that an app requesting broad contact access be reviewed to check whether that access is actually required for its stated function. This remediation text is attached to its corresponding finding in the final report, so the report tells the user not just that something is risky, but what a reasonable next step looks like.

III. MODELING AND ANALYSIS

The system is organised as a three-layer architecture rather than a heavier design such as MVC or a full service-oriented layout, mainly because a flat, layered structure was easier to build, test and extend within the scope of a single academic project. The presentation layer is the web interface where a user uploads an APK and later views the generated report. The application layer sits behind it and does all of the actual work, file validation, decompilation, static and dynamic analysis, risk scoring and report assembly. The data layer stores uploaded files, extracted permissions, runtime

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 2, July 2026
Impact Factor: 8.2

observations, computed scores and generated reports, and also keeps a scan history so that a previously analysed APK does not need to be re-scanned from scratch.

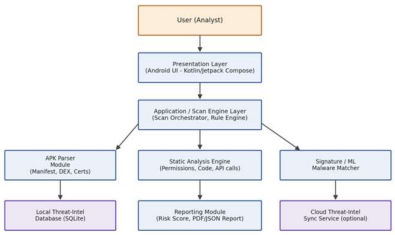

Fig. 1. Layered system architecture of APK Sentinel.

IV. RESULTS AND DISCUSSION

The prototype was built with Python and Flask on the backend, React.js on the frontend, and Apktool and an Android emulator for the static and dynamic analysis stages respectively; SQLite was used during development with a path to MySQL for larger deployments. To validate the system, a mixed set of APK samples, a small number of clean, benign applications alongside known-malicious samples drawn from sources such as Drebin and AndroZoo, were run through the full pipeline, and the resulting risk classifications were cross-checked against MobSF, an established open-source mobile security framework, to confirm that the permissions, manifest findings and component exposures identified were consistent between the two tools.

Table I summarises a representative subset of the test cases used during system and validation testing. Test cases covering a benign application, an application requesting a combination of SMS and overlay permissions, and an application matching a known malware signature were all classified at the expected risk level, and the system additionally handled a corrupted APK gracefully rather than failing silently, and produced a valid PDF export of the generated report.

TABLE I. Representative System and Validation Test Cases

TC ID	Description	Expected	Actual	Status
TC01	Benign APK, minimal permissions	Safe (<35)	12, Safe	Pass
TC02	SMS + overlay permissions	Suspicious/Malicious	78, Malicious	Pass
TC03	Known malware signature	Malicious	95, Malicious	Pass
TC04	Corrupted/invalid APK	Graceful error	Error dialog	Pass
TC05	Export report as PDF	Valid PDF	PDF generated	Pass

DOI: 10.48175/IJARSCT-37756

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 2, July 2026
Impact Factor: 8.2

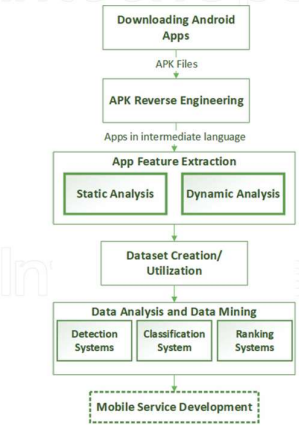

Fig. 2. Data flow through the static and dynamic analysis pipeline.

Overall accuracy of the manifest-level findings tracked closely with MobSF across the sample set, which was expected since both tools ultimately draw on the same AndroidManifest.xml data. The main practical difference observed during testing was not in what was detected but in how it was communicated: MobSF's report format assumes a reader already fluent in Android permission taxonomy, whereas APK Sentinel's report translates the same findings into short, specific sentences (for example, explaining what a specific exposed activity could allow another app to do), and pairs each one with the LLM-generated remediation suggestion described in Section II-C. This confirms the underlying premise of the project, that the technical accuracy achievable with static and dynamic analysis does not by itself guarantee that a user can act on the result, and that the presentation layer is doing at least as much work as the analysis engine itself in making the system useful outside a security team.

Dynamic analysis did add real value in a small number of cases where an app's declared permissions understated what it actually did at runtime, though the fixed observation window used during testing means that behaviour triggered only after a longer delay, or only in response to a specific user interaction, would not reliably be caught. This is a known limitation of time-boxed sandbox analysis generally and is discussed further as a direction for future work.

V. CONCLUSION

This paper described APK Sentinel, a system that combines static manifest analysis and sandboxed dynamic analysis of Android APK files into a single, weighted risk score, and then explains that score using plain language rather than raw technical output. Static analysis surfaces what an application asks for and what it exposes; dynamic analysis checks a sample of what it actually does once running; and the two are deliberately kept in one pipeline so a user receives a single answer rather than two reports to reconcile on their own. Validation against benign and known-malicious samples, cross-checked with MobSF, showed that the underlying detection was consistent with an established tool, while user-facing testing indicated that the plain-language report format made the result considerably easier to act on for someone without a security background.

The current implementation still has clear limits, the risk score is computed with a fixed, rule-based weighting rather than a learned model, the dynamic analysis window is short, and the system is scoped to Android APK files only. These are the natural next steps: incorporating a machine-learning classifier (for example, a Random Forest model trained on labelled permission and behaviour data) to refine the risk score, adding signature-based matching against known malware databases for faster triage, extending the sandbox observation period and interaction depth, and eventually generalising the same approach to other packaging formats such as iOS IPA files. A community-sourced threat intelligence layer, where users can anonymously contribute scan results, is also a reasonable long-term direction once the core pipeline is proven at scale.

DOI: 10.48175/IJARSCT-37756

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 2, July 2026
Impact Factor: 8.2

ACKNOWLEDGMENT

The authors would like to thank the Department of MCA, Maharaja Institute of Technology Mysore, for the guidance and infrastructure support extended during the course of this project.

REFERENCES

[1] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android Security," IEEE Security & Privacy, vol. 7, no. 1, pp. 50-57, 2009.
[2] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in Proc. IEEE Symp. Security and Privacy, 2012, pp. 95-109.
[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in Proc. NDSS Symp., 2014.
[4] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android Permissions Demystified," in Proc. ACM CCS, 2011, pp. 627-638.
[5] P. Faruki et al., "Android Security: A Survey of Issues, Malware Penetration, and Defenses," IEEE Commun. Surveys Tuts., vol. 17, no. 2, pp. 998-1022, 2015.
[6] A. F. A. Kabakus, I. A. Dogru, and A. Cetin, "APK Auditor: Permission-based Android Malware Detection System," J. Netw. Comput. Appl., vol. 52, pp. 145-154, 2015.
[7] J. M. Arif, M. F. Ab Razak, S. Awang, S. R. T. Mat, N. S. N. Ismail, and A. Firdaus, "A Static Analysis Approach for Android Permission-Based Malware Detection Systems," PLOS ONE, vol. 16, no. 9, 2021.
[8] K. Allix, T. F. Bissyande, J. Klein, and Y. Le Traon, "AndroZoo: Collecting Millions of Android Apps for the Research Community," in Proc. MSR, 2016, pp. 468-471.
[9] OWASP Foundation, "OWASP Mobile Application Security Verification Standard (MASVS)," owasp.org, accessed 2026.
[10] Google, "Android Developers - App Manifest Overview," Android Open Source Project Documentation, developer.android.com, accessed 2026