أكاديمية كاوست
KAUST ACADEMY

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

AI-Driven APK Threat Analysis & Reporting

Turki Algethami, Ahmed Alghamdi, Talal Alamri,
Saud Alasmari, Hashim Aljafri, Muhammad Ashi,
Prof. Ali Hassan

turki.algethami@kaust.edu.sa, ahmed.alghamdi@kaust.edu.sa, talal.alamri@kaust.edu.sa,
saud.alasmari@kaust.edu.sa, hashim.aljafri@kaust.edu.sa, muhammad.ashi@kaust.edu.sa,
ali.hassan.1@kaust.edu.sa

Abstract

Android's open application model allows software to be installed from outside the official Play Store. Sideloaded and third-party APK files are convenient, but they are also one of the most significant malware entry points on the platform, precisely because they bypass the Play Store's vetting. This project develops an AI-powered platform that performs deep static analysis on APK files *before* installation, systematically inspecting decompiled application code for embedded malware, obfuscated payloads, and unauthorized data-access behaviours. Rather than stopping at raw forensic output, every finding is passed to a language model that distills the technical results into a single calibrated risk score and a clear, plain-language report, so that a non-technical user can make an informed decision about whether an application is safe to install. Internally, the platform is organized as an adaptive, tiered pipeline: the cheapest checks run first (internal cache lookup, hash reputation lookup, static analysis), and expensive sandboxed dynamic analysis is invoked only when the static evidence is genuinely inconclusive. Two scan modes are offered — a Quick Scan and a Deep Scan — and results are delivered through a full-stack Flask/React security dashboard as an interactive experience rather than a raw scan log.

Date: August 24, 2026
Code: https://github.com/MomentousBruh/apk-scanner

Contents

1 Introduction **2**

2 Problem **2**

2.1 Who Is Affected . . . 2

2.2 The Deeper Problem . . . 2

2.3 Problem Statement . . . 3

3 Related Work **3**

3.1 Comparison of Available Tools . . . 4

3.2 Summary of the Gap . . . 4

3.3 The Closest Comparison . . . 4

4 Methodology **5**

4.1 Method details . . . 5

4.2 Evaluation Procedure . . . 7

4.3 Implementation Progress . . . 7

5 Results **15**

5.1 Evaluation Design . . . 15

5.2 Experiment 1: Complete Pipeline . . . 15

5.3 Experiment 2: Detection Without Reputation Data . . . 16

5.4 Where the Detections Come From . . . 17

5.5 Results by Malware Family . . . 18

5.6 Scope of the Evaluation . . . 19

6 Discussion **19**

6.1 The Platform Detects Malware Without Reputation Data . . . 19

6.2 The Sandbox Recovers What the Classifier Misses . . . 20

6.3 Obfuscation Is Not What Causes Misses . . . 20

6.4 Hydra Is a Family-Specific Failure . . . 20

6.5 Detection Versus Decisiveness . . . 21

6.6 Where the Results Meet and Miss Expectations . . . 21

7 Limitations and Future Work **21**

7.1 Sandbox Evasion . . . 21

7.2 External Dependencies . . . 21

7.3 Reduce External Dependency . . . 21

8 Conclusion **22**

References **22**

1 Introduction

Android's open application model lets users install software from outside the official Play Store. Sideloaded and third-party APK files are convenient, but they are also one of the most significant malware entry points on the platform precisely because they bypass the Play Store's vetting. Once such an app is installed, any malicious behaviour it contains has already had the opportunity to run.

This project develops an AI-powered platform that performs deep static analysis on APK files *before* installation. The system systematically inspects decompiled application code to detect embedded malware, obfuscated payloads, and unauthorized data-access behaviours. Rather than stopping at raw forensic output, every finding is passed to an AI model that distills the technical results into a single, calibrated risk score together with a clear, plain-language report. The goal is to let a non-technical user make a confident, informed decision about whether an application is safe to install.

Internally, the platform is organized as an adaptive, tiered analysis pipeline. The cheapest and fastest checks run first (an internal cache of previously seen files, hash reputation lookups, and static analysis), and expensive sandboxed dynamic analysis is invoked only when the static evidence is genuinely inconclusive. Two scan modes are offered: a **Quick Scan** (hash reputation check, internal repository lookup, static analysis, static ML verdict, and an LLM-generated report) and a **Deep Scan** (everything in Quick Scan plus sandboxed dynamic analysis when static analysis is inconclusive, followed by a final dynamic-aware report). The result is delivered through a full-stack Flask/React security dashboard as an interactive experience rather than a raw scan log.

2 Problem

2.1 Who Is Affected

Sideloaded and third-party APKs create risk for two very different groups, each of which the current tooling fails in a different way:

- **Non-technical users** have no reliable way to judge whether an APK is safe before installing it. The warnings they do see (permission prompts and Play Protect notices) are shallow and are rarely explained in terms a non-specialist can act on.
- **Security analysts and reverse engineers** *can* detect threats manually, by decompiling with `apktool` [1]/`jadx`, reading smali, and inspecting permissions. But this work is slow, requires deep expertise, and does not scale to the volume of apps circulating outside the official stores.

2.2 The Deeper Problem

At the level of individual techniques, Android malware detection is a mature and well-studied field: static feature extraction, dynamic sandboxing, ML classification, and reputation lookups all already exist and work. So the problem this project targets is not "how do you detect Android malware."

There are, in fact, two distinct gaps. The first is one of *accessibility*: analyst-grade output (CWE codes, exported components, obfuscation patterns) is technically rich but unreadable to the very users who most need protection, while consumer-grade tools give a verdict with no reasoning. No mainstream tool uses an LLM to translate deep static-analysis output into a risk score *and* a clear explanation before installation.

The second gap is *operational*. Because dynamic analysis is expensive, slow, and hard to scale, a practical system must decide, automatically and defensibly, which of the cheaper checks are

sufficient and when the expensive check is actually justified. This is a resource-allocation and decision-policy problem layered on top of existing detection techniques, not a new detection technique. Existing open tools run every stage on every file regardless of need, and none expose a documented policy for *when not to bother* with the costly step.

2.3 Problem Statement

Problem Statement

There is no accessible tool that combines deep static analysis of APK internals with AI-generated, plain-language risk explanations, thereby bridging the gap between raw reverse-engineering output and a decision a non-expert user can actually act on, while also deciding, in a cost-aware and defensible way, how much analysis each file actually warrants before it is installed.

3 Related Work

Several tools cover parts of this space. None combine all three of (a) deep static decompilation, (b) automated malware/behaviour detection, and (c) an AI layer that turns findings into a plain-language verdict for non-experts, and none pair that with a cost-aware decision layer that avoids unnecessary analysis.

3.1 Comparison of Available Tools

Table 1 Existing tools covering part of this space, and the gap each one leaves.

Tool	What it does (strengths)	Gap / weaknesses vs. our project
MobSF (Mobile Security Framework) [2]	Open-source static + dynamic analysis for APK/IPA. Detects malware patterns, bad permissions, weak crypto, and hardcoded secrets. Offers a web dashboard, REST API, and a VirusTotal lookup add-on.	Built for security engineers, not end users. Raw findings (CWE codes, permission lists) need expertise to interpret; no plain-language risk verdict and no pre-install on-device use. Runs static *and* dynamic analysis unconditionally on every submission, with no reputation pre-filter, no cache, and no decision layer for whether deeper analysis is even needed.
VirusTotal (mobile scanning) [3]	Multi-engine signature/heuristic scan of an uploaded APK; aggregates AV vendor verdicts, and answers instantly for any file it has already seen.	Signature-based and therefore weak against new or obfuscated malware. No code-level static analysis and no human-readable reasoning; only a vendor score list.
apktool-mcp-server (zinja-coder) [4]	An MCP bridge that lets an LLM (e.g. Claude) call `apktool` to decode/rebuild APKs and read manifest/resources during a chat session.	A developer tool: manual setup, with no built-in malware detection, scoring, or report generation; it requires the user to already know what to look for. Our platform performs its own decoding and adds the detection and AI-reporting layers that this tool leaves to the user. We plan to use its decoding layer as one building block and add the detection and AI-reporting layers on top.
Google Play Protect [5]	Built-in on-device scanning for apps from the Play Store and for sideloaded APKs; blocks known malware.	Closed and non-transparent, and weaker on sideloaded/unknown apps. Provides no detailed report or reasoning for the user.
Koodous [6]	Community-driven Android malware analysis platform with YARA rule scanning and crowd-sourced verdicts.	Depends on community rule coverage. Verdict-only output with minimal explanation for non-technical users.

3.2 Summary of the Gap

The landscape splits cleanly into two camps, and our project sits in the space neither one occupies:

- **Analyst-grade tools** (MobSF, `apktool`/`jadx`-based tools) produce technically rich but jargon-heavy output aimed at people who already know what CWEs, exported components, and obfuscation patterns mean.
- **Consumer-grade tools** (Play Protect, AV scanners) give a simple verdict but little to no reasoning, and are weak against novel or obfuscated threats because they lean on signatures.
- **No mainstream tool** uses an LLM to translate deep static-analysis output into a risk score *plus* a clear explanation before the user installs the app.

3.3 The Closest Comparison

The closest existing point of comparison is **MobSF** [2]: a mature open-source tool that performs both static and dynamic analysis of APKs in a single dashboard. Crucially, however, MobSF is a *framework built for technical users*, security engineers and reverse engineers who already know how to read its raw findings. Our project instead targets an easy-to-use, non-technical friendly, and

efficient platform that turns the same class of analysis into a plain-language verdict an ordinary user can act on.

The two also differ in how the analysis is run. MobSF executes static and dynamic analysis unconditionally on every submission, with no reputation pre-filter, no cache of previously analyzed files, and no decision layer that judges whether dynamic analysis is even necessary. In short, MobSF answers "what does this APK do," while this project answers "should we even bother finding out, and how much checking is actually warranted," and delivers that answer in a form a non-expert can understand.

4 Methodology

4.1 Method details

4.1.1 Overview

Our platform lets a user submit an APK file for analysis through a simple web interface, with two scan modes: a **Quick Scan** (static analysis) and a **Deep Scan** (static + dynamic analysis). From the moment the file is submitted, the entire process runs *without any user interaction*: the platform performs the analysis end-to-end and returns a single, jargon-free report together with a risk assessment that a non-technical user can act on directly.

The pipeline is designed to be efficient as well as accessible: expensive analysis runs only when no cheaper source can already answer the question.

4.1.2 Analysis Flow

When an APK is submitted, the backend processes it in the following order:

1. **Internal cache check.** The backend computes the hash of the APK and first checks our own database. If this exact file has already been analyzed, for example, uploaded earlier by a different user, the stored report is returned instantly, with no external API call and no re-analysis.
2. **VirusTotal reputation check.** If the file is not in our database, the backend queries the VirusTotal API by hash. If VirusTotal already knows the file, its information is sent directly to the report-generation model, skipping local analysis entirely.
3. **Decompilation.** If the file is unknown to both sources, it is processed on our server: the decompilation component decompiles the APK and forwards the decompiled application with its extracted information to the analysis model.
4. **Analysis - ML Model.** A pre-trained ML Model analyzes the decompiled APK, examining its contents for embedded malware, obfuscated payloads, and unauthorized data-access behaviours, and passes its findings to the report-generation model.
5. **Dynamic analysis — sandbox.** On a Deep Scan, if the static evidence is inconclusive or the application is flagged as obfuscated, the sample is submitted to the Tria.ge sandbox for instrumented execution, and the resulting behavioural report is passed to the report-generation model alongside the static findings. On a Quick Scan, or where the static evidence is already conclusive, this step is skipped.
6. **Report generation - LLM.** A language model converts the analysis results (from the ML model or from VirusTotal) into an easy-to-read, non-technical, and precise report, and returns it to the backend.
7. **Store and return.** The backend saves the APK hash together with the generated report in the database. Therefore, any future submission of the same file is served instantly from step 1,

and returns the report to the user.

The deliberate ordering of steps 1–4 means each stage acts as a filter for the next: a database lookup is free, a VirusTotal hash lookup is cheap, and full decompilation plus ML analysis is reserved only for files that neither source has seen before.

4.1.3 System Architecture

Figure 1 shows the overall architecture and the flow of an APK through the platform.

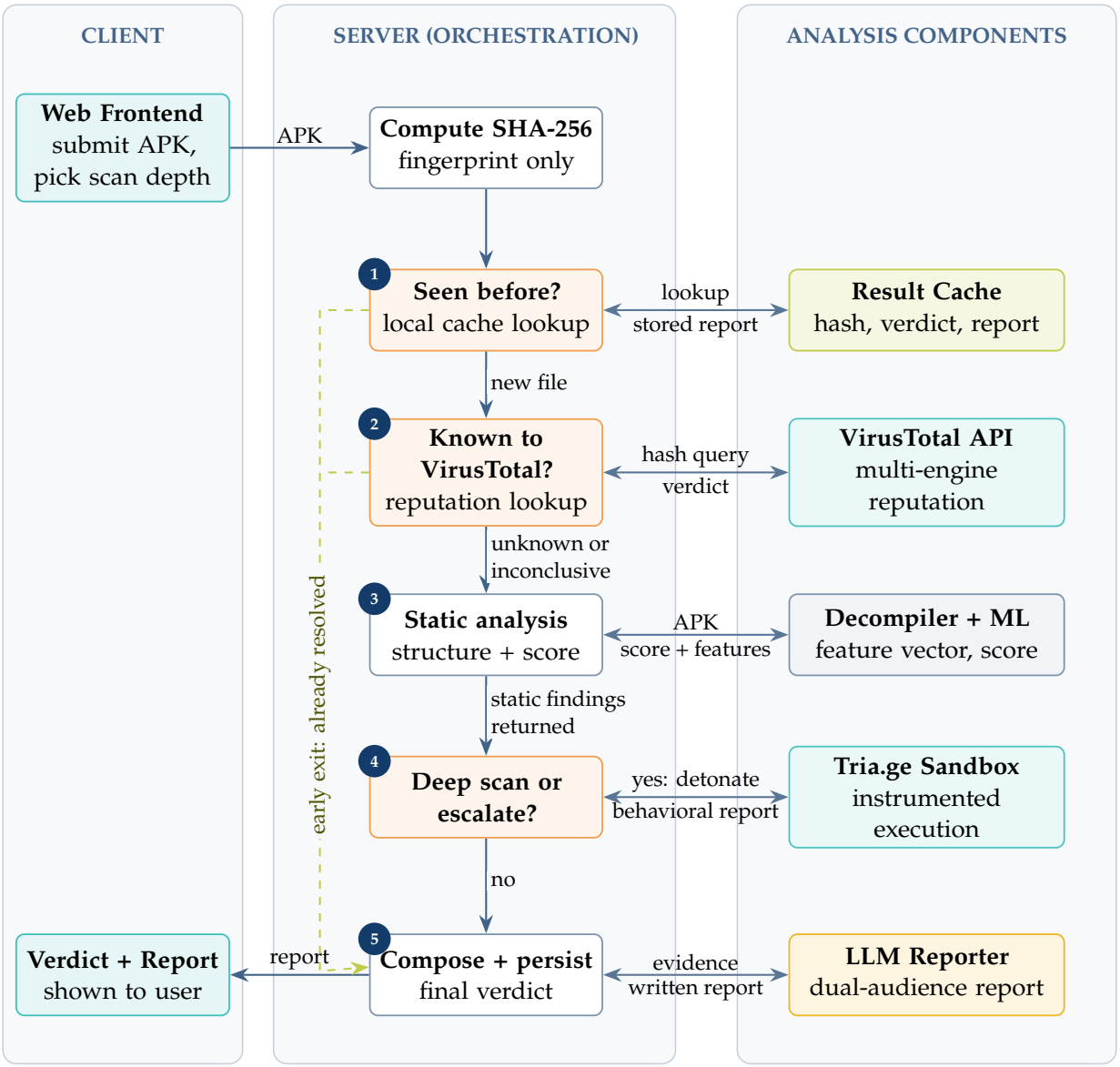


Figure 1 Platform architecture. The server lane owns every decision; analysis components are invoked only when the preceding, cheaper stage cannot resolve the file. Dashed paths are early exits that skip the remaining analysis.

4.2 Evaluation Procedure

4.2.1 Datasets

- **LAMDA** – used to train and evaluate the *static* ML model. LAMDA is the largest longitudinal Android malware benchmark to date: over 1 million APKs spanning 1,380 malware families and 13 years (2013–2025), with static features derived from the Drebin feature set (permissions, intent filters, exported components, API calls).
- **AndroZoo Samples Database** – AndroZoo is a growing collection of Android Applications collected from several sources, including the official Google Play app market. Current number of APKs: 27,578,851.[7]

4.2.2 Tools, Libraries, and Frameworks

- **Tria.ge (Cloud-Sandbox)** – A cloud-based isolated sandbox that runs samples against pre-defined rules.
- **SQLite** – chosen as the platform's database for the hash/report cache.
- **Flask (backend) / React (frontend)** – backend orchestrates the pipeline; frontend provides upload and scan-mode selection.

4.3 Implementation Progress

4.3.1 Platform

The platform follows a client-server separation. A browser-based interface handles submission and presentation, while an application server exposes a small set of endpoints that coordinate the analysis stages and return their results.

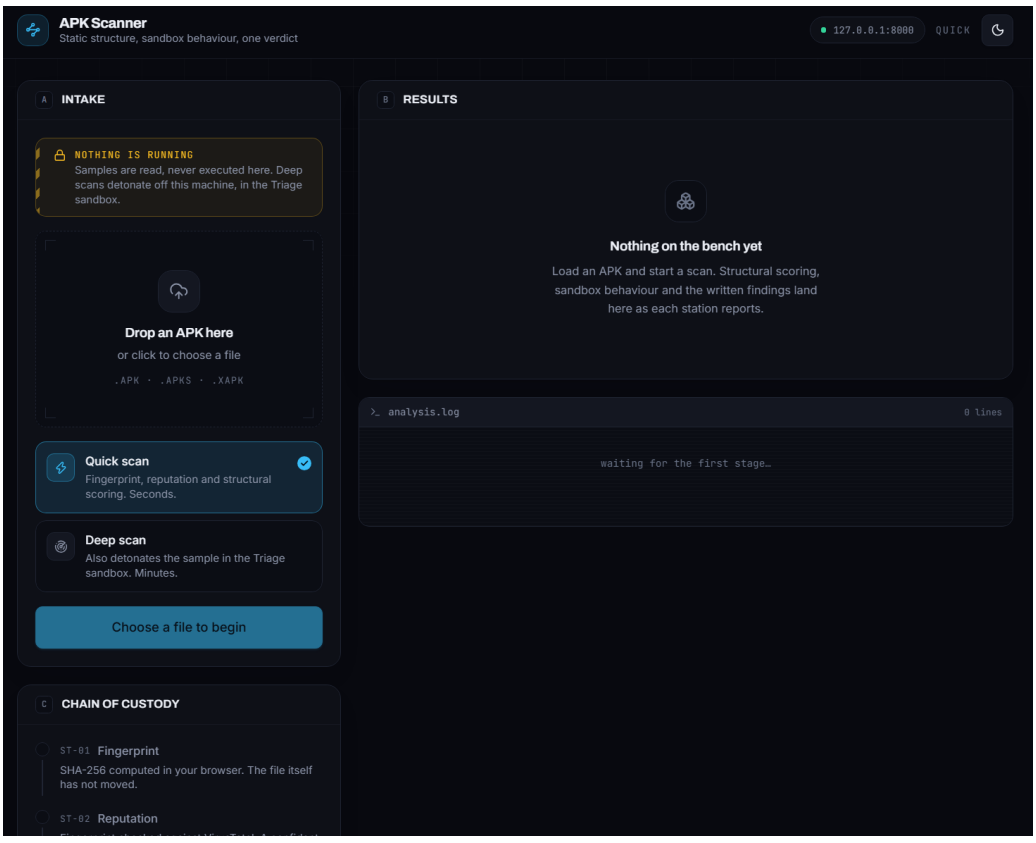


Figure 2 Platform User-Interface

Submission workflow. A deliberate design decision governs the order of operations. When a user selects an application package, its SHA-256 cryptographic fingerprint is computed locally in the browser, and only this fingerprint is transmitted for an initial lookup. The file itself is uploaded only when the fingerprint alone cannot resolve the question. This ordering yields three benefits: previously analysed files return an answer in a single lightweight request; unnecessary transfer of large binaries is avoided; and a user whose file is already known never has to upload it at all, which is meaningful when the submitted application may contain personal data.

Two scan depths are exposed. A *quick* scan performs reputation and static structural analysis and completes in seconds. A *deep* scan additionally executes the application inside an instrumented sandbox and generates a full written report, a process measured in minutes rather than seconds.

Progressive result streaming. Because analysis duration varies by two orders of magnitude between the two modes, results are streamed incrementally rather than returned as a single response. The server emits a sequence of progress events describing the stage currently executing, a completion percentage, and a human-readable description, followed by a terminal event carrying the verdict. The interface renders these as a chain-of-custody sequence in which each analysis stage is shown as a discrete station that advances from pending, to active, to cleared. This makes a multi-minute analysis legible to a non-technical user, who can see what is happening and why it is taking time.

A practical complication arose in reconciling the two analysis pipelines, each of which reports its own progress on an independent zero-to-one-hundred scale. Presented naively, the indicator would advance to near completion, reset, and advance again. Each pipeline's progress is therefore rescaled into a disjoint segment of a single monotonic track, so that the reported figure never moves backwards.

Reputation integration. Before any local analysis is performed, the submitted fingerprint is checked against a large multi-engine reputation service. The response is deliberately mapped to three distinct states rather than a single score, because the distinction between them carries different meaning for the workflow:

- **Unknown.** The file has never been observed. This is treated as an absence of evidence rather than evidence of safety, and analysis proceeds to the subsequent stages.
- **Conclusive.** A clear consensus exists in either direction. The result is returned immediately, the file is never uploaded, and no further analysis is performed.
- **Inconclusive.** A minority of engines disagree with the majority. This represents genuine ambiguity rather than silence, and is precisely the case in which local analysis adds the most value, so the workflow escalates.

The separation of *unknown* from *inconclusive* is important. Both would superficially appear as a non-answer, but they warrant different treatment: the former reflects a file that no one has encountered, while the latter reflects a file that has been examined and produced disagreement.

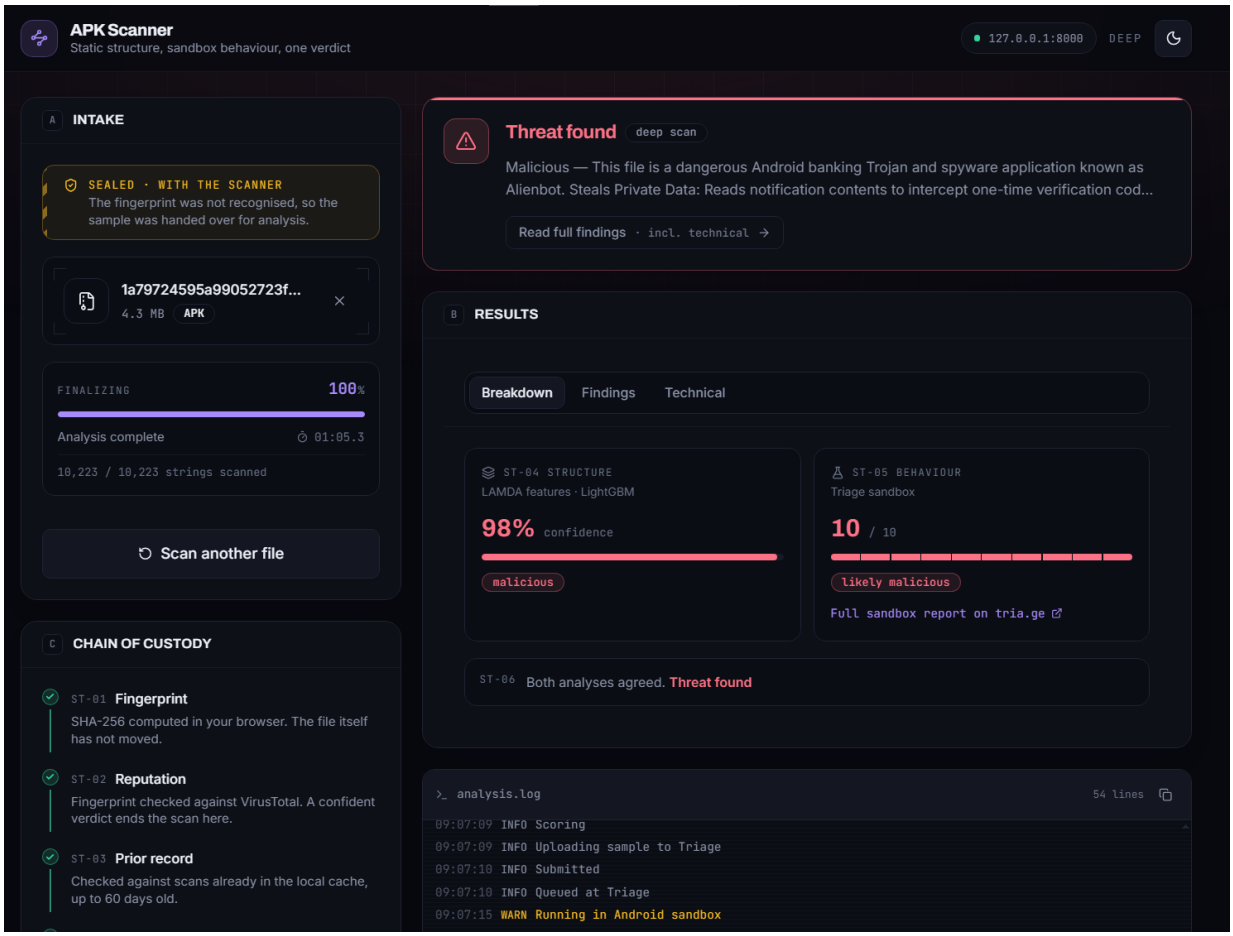


Figure 3 Results of Scanning

Persistence and caching. Analysis results are persisted in a relational store keyed by the file fingerprint. Each record holds the final verdict, the numeric scores from each contributing stage, the generated written report, and a reference to the behavioural analysis run. Records carry an analysis timestamp and are treated as valid for a bounded freshness window, after which a resubmitted file is analysed afresh rather than served from cache. This bound reflects the fact that a verdict is a statement about knowledge at a point in time: reputation data improves, detection capability advances, and an old benign verdict is weaker evidence than a recent one.

Writes use insert-or-update semantics so that a file re-analysed at greater depth enriches its existing record rather than creating a duplicate. A file first examined by quick scan and later by deep scan therefore accumulates behavioural results alongside its original structural findings. The write path validates supplied fields against the schema and discards unrecognised ones, so that extending the analysis pipeline with new outputs cannot cause a successful analysis to be lost at the persistence step.

4.3.2 Static Analysis

The static stage characterises an application without executing it, by examining its declared structure and its compiled code.

Feature representation. An application package is reduced to a fixed-length binary vector, where each position records the presence or absence of one specific observable trait. The trait vocabulary is fixed in advance and derives from a large-scale labelled corpus of Android applications,

ensuring that every analysed sample is described in the same terms and that the representation is directly comparable across submissions. Traits are drawn from two distinct evidence layers:

- **Declarative layer.** The application manifest yields requested permissions, declared components such as activities, background services, broadcast receivers and content providers, registered intent filters, and required hardware capabilities. This describes what the application announces it will do.
- **Implementation layer.** The compiled bytecode yields the security-relevant platform interfaces the application actually invokes, together with network endpoints embedded as literal strings. This describes what the application is equipped to do.

Separating these layers is analytically useful, since divergence between them is itself informative: an application whose code reaches for capabilities its manifest does not justify is more interesting than one where the two agree.

Extraction strategy. A naive implementation would enumerate every method signature and every string constant in an application, accumulate them, and intersect the result with the trait vocabulary. For large applications this produces intermediate collections of tens of thousands of entries, the overwhelming majority of which are discarded.

The implemented approach inverts this. A lookup structure mapping each trait to its position in the output vector is constructed once when the schema is loaded. During extraction the application is traversed a single time, and each observed method signature or embedded endpoint is tested directly against this structure, with the corresponding vector position set immediately on a match and the observation discarded otherwise. Non-matching observations are never retained. A further reduction follows from the observation that the analysis requires only the names of invoked interfaces and the literal strings present, never the relationships between callers and callees; the construction of a cross-reference graph, ordinarily the dominant cost in bytecode analysis, is therefore avoided entirely. Traversal also emits incremental progress, allowing the interface to report meaningful advancement on large applications rather than appearing stalled.

Classification. The feature vector is scored by a gradient-boosted decision tree ensemble trained on the same labelled corpus from which the trait vocabulary derives. This model family was selected because the input is high-dimensional, sparse and binary, a regime in which tree ensembles are effective, and because the learned structure permits per-trait importance to be recovered. That property is used directly in report generation, where the traits contributing most strongly to a decision are surfaced as evidence.

Decision policy. The continuous model output is mapped onto a three-zone policy rather than a simple threshold. Scores above an upper bound of 0.90 are reported as malicious; scores below a lower bound of 0.05 are reported as benign; everything between is reported as inconclusive and escalated for deeper examination. The two bounds are set conservatively, leaving a wide middle zone, which reflects an explicit position on the relative cost of the two error types. Wrongly clearing a malicious application exposes a user to real harm, whereas wrongly flagging a benign one causes inconvenience and can be corrected by further analysis. The middle zone is therefore deliberately wide, and the system is designed to answer "not certain, examine further" rather than to force a binary decision it cannot support.

Obfuscation as an independent signal. Applications are separately examined for signatures of packing and obfuscation. This result is not treated as evidence of malice, since such protections have entirely legitimate commercial uses. It is instead treated as a statement about the reliability

of the static analysis itself: where an application's true logic is concealed until runtime, the structural evidence available to the classifier is incomplete, and a low score reflects the absence of visible indicators rather than their genuine absence.

Consequently, an application identified as obfuscated cannot be cleared on static evidence alone regardless of how benign its score appears, and is escalated to behavioural analysis. Treating a confidently low score on an obfuscated file as a clean result would invert the meaning of the evidence. This detection is also implemented to fail safe: if it cannot be performed, the analysis proceeds without it rather than aborting, since it is a supporting signal and not a precondition.

4.3.3 Dynamic Analysis

Static analysis cannot observe behaviour that materialises only at runtime, including code retrieved after installation, logic gated behind runtime conditions, and network communication with remote infrastructure. The dynamic stage addresses this by executing the application in an isolated, instrumented environment and recording what it does. In this type of analysis, *Tria.ge* has been utilized as the primary sandboxing platform. Following approval for a researcher account, the platform provides the ability to integrate custom YARA rules into the analysis workflow. Leveraging this capability enhances the detection effectiveness of the *Tria.ge* sandbox, particularly when analyzing malware that employs advanced evasion techniques such as packing and obfuscation.

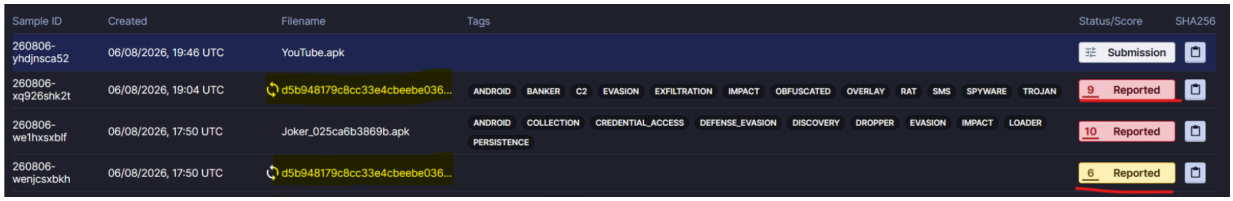


Figure 4 Results of Analysis After rules enhancements

Execution and observation. On a deep scan, the application is submitted to a controlled sandbox environment where it is installed and executed on an instrumented Android system for a bounded interval. Instrumentation records network activity including name resolution and contacted endpoints, file system activity including files created and modified, process and service activity, and security-relevant platform calls. The environment is fully isolated, so execution of hostile code carries no risk to the analysis infrastructure or to the user.

Execution is intentionally performed remotely in a purpose-built environment rather than locally. Maintaining an instrumented sandbox that is both convincing enough to elicit genuine malicious behaviour and hardened enough to contain it is a substantial undertaking in its own right, and delegating it allows the project to concentrate on analysis and interpretation.

Lifecycle management. A behavioural run is asynchronous and passes through queueing, scheduling, execution and post-processing before results become available. The system tracks this lifecycle by polling and maps each observed state onto the unified progress track presented to the user, interpolating within the execution phase using elapsed time against the configured run duration so that the indicator advances continuously rather than in discrete jumps. Bounded waiting is enforced: a run that does not complete within its expected duration plus a tolerance is abandoned, and the scan falls back to its static result rather than leaving the user waiting indefinitely.

Multiple execution environments. An application may be executed on more than one system image, since behaviour can be conditional on platform version or device characteristics. Where multiple runs are produced, the run exhibiting the most severe behaviour is taken as authoritative. The reasoning is that malicious behaviour observed in any environment is real behaviour, whereas its absence in another environment may simply indicate that the relevant precondition was not met. Selecting the most severe outcome is the position consistent with treating absence of evidence as weaker than evidence of presence. Individual runs are preserved unmodified rather than merged, so that the selected result remains a faithful account of a single execution rather than a synthesis that never occurred.

Verdict integration. The behavioural stage produces a severity rating on its own scale, which is normalised into the same four-value vocabulary used throughout the system, comprising benign, suspicious, malicious and inconclusive. This shared vocabulary is what makes integration across heterogeneous stages possible at all: reputation data, a probabilistic classifier and a behavioural rating are otherwise incommensurable.

Integration follows a severity-dominant rule. Where both the static and behavioural stages reach a conclusion, the more severe of the two prevails; where either is inconclusive, it defers to the other rather than diluting it. The justification is that both stages observe genuinely different evidence, and neither is a strict refinement of the other: behavioural analysis can reveal malice invisible in structure, while structural analysis can reveal capability that a time-bounded execution did not happen to exercise. A finding by either is therefore treated as a finding.

Failure isolation. By the time behavioural analysis begins, a complete static verdict already exists. The dynamic stage is consequently implemented as an enrichment rather than a dependency: if the sandbox is unreachable, the run fails, or execution produces no observable behaviour, the scan returns its static verdict together with an explicit statement that behavioural analysis could not be completed. A deep scan therefore never returns less information than a quick scan would have, and the user is told which evidence the verdict does and does not rest upon.

4.3.4 Report Generation

The preceding stages produce a verdict and a large volume of structured technical evidence. Neither is directly useful to the non-technical user the system is intended to serve. The final stage converts this evidence into written explanation using a large language model, Gemini, accessed through its hosted API.

Position in the pipeline. The language model is placed strictly downstream of the decision. It does not classify, does not score, and does not influence the verdict; it receives an already-determined result together with the evidence supporting it, and produces prose. This separation is deliberate and central to the design. The verdict remains fully deterministic, reproducible and attributable to specific observed evidence, while the language model is confined to the task it is genuinely reliable at, namely rendering technical material in accessible language. A generative component permitted to influence the verdict would make the system's decisions neither reproducible nor auditable.

Input preparation. Raw behavioural output is verbose, deeply nested, and highly variable in size. Submitting it unmodified would be slow and costly, and risks the salient findings being diluted within routine detail. A condensation step therefore bounds the input along three axes: nesting depth, the number of elements retained from long sequences, and the length of individual text values.

This condensation is deliberately structure-agnostic. It applies generic bounds rather than selecting known fields by name, so that changes to the upstream output format degrade the report's richness rather than breaking generation outright. Where content is elided, an explicit marker records that truncation occurred, so the model is aware that its view is partial and can say so rather than presenting an incomplete picture as complete.

Dual-audience generation. A single inference produces two distinct reports describing the same finding for two different readerships. The non-technical report addresses a general reader and follows a fixed sequence: the verdict and its reason, a plain description of observed behaviour, the practical significance of that behaviour, the most important supporting evidence, concrete recommended actions, and a statement of confidence and limitations. The technical report addresses a security practitioner and covers classification, behavioural detail, indicators of compromise, mapping to a standard adversary technique taxonomy, containment measures, and an explicit account of gaps in the underlying data.

Generating both in one inference rather than two was a considered choice. It halves latency and cost, and more importantly guarantees that the two reports describe the same evidence in the same terms; independent generations could diverge in emphasis or conclusion, which would be actively misleading when the same user may read both. The two sections are separated by unambiguous delimiters and recovered programmatically after generation.

Grounding constraints. The most significant risk in this stage is fabrication, since a fluent but invented indicator is more damaging than no indicator at all: it may be acted upon. Several constraints address this. The model is instructed to base every claim strictly on the supplied evidence and to state plainly when information is missing rather than inferring plausible substitutes. Indicators such as network endpoints and file paths must be reproduced exactly rather than paraphrased, since an approximated indicator is useless operationally. Absent categories of evidence must be declared explicitly rather than omitted silently, so that a gap in the data is visible to the reader instead of being mistaken for a clean result. Generation parameters are set for determinism and consistency rather than variety, and the reasoning budget is constrained so that it cannot displace the visible output.

Validation and failure handling. Generated output is validated for structural completeness before acceptance, and incomplete generation is detected and reported rather than surfaced as a truncated report. As with the behavioural stage, failure is isolated: if report generation cannot be completed, the verdict, the numeric scores and the extracted indicators are still returned, accompanied by a statement that the written explanation was unavailable. The analytical result never depends on the availability of the component that describes it.

Presentation. Generated reports are structured documents rather than plain prose, and are rendered accordingly in the interface, with headings, evidence lists and verbatim indicators presented distinctly. The two audience versions are offered as alternative views of the same finding, so that a general reader is not confronted with technical material while a practitioner retains access to the full detail. Rendering treats all generated text as untrusted content: because reports quote strings originating from the analysed application, including file paths and network endpoints chosen by its author, quoted material is displayed strictly as literal text and can never be interpreted as instructions by the presentation layer.

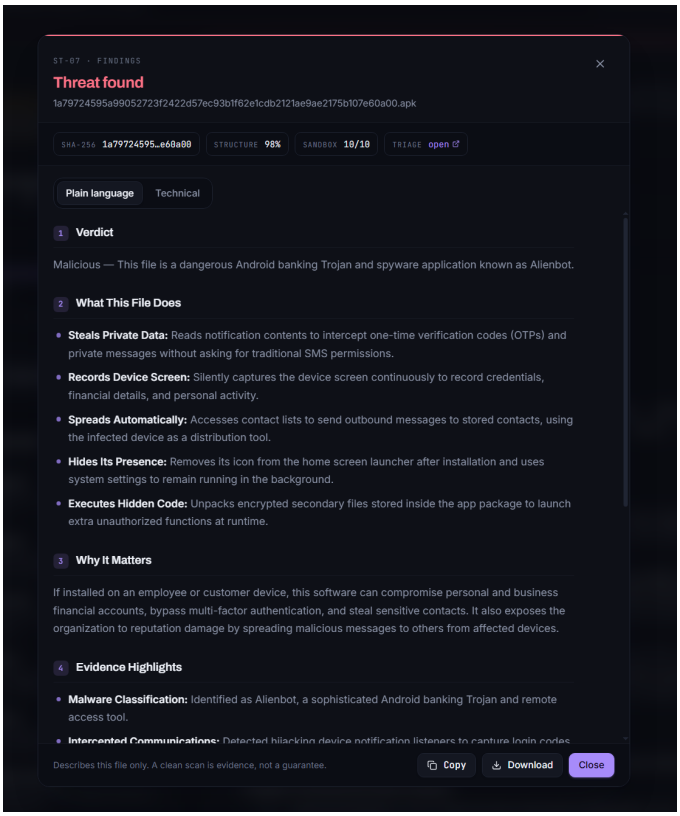


Figure 5 Summary Report For Non-Technical Users

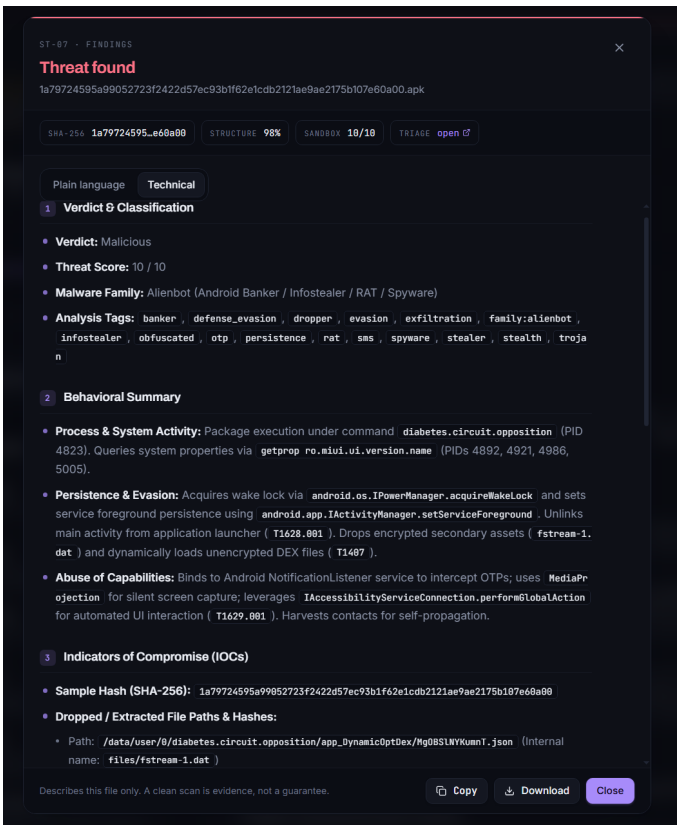


Figure 6 Report For Technical Experts

5 Results

5.1 Evaluation Design

The system was evaluated in two experiments, each answering a different question.

Experiment 1 measures the complete pipeline as deployed, with the VirusTotal reputation lookup enabled. It uses 968 malicious packages drawn from AndroZoo and 1,116 benign packages, of which 1,056 are AndroZoo applications labelled benign and 60 come from F-Droid, a curated repository of open-source Android software. Including a benign corpus allows both error types to be measured: how often malware is cleared, and how often harmless software is flagged.

Experiment 2 removes the reputation lookup entirely. Every sample is processed by static analysis and sandboxed execution alone, with no VirusTotal query at any point. This experiment exists because Experiment 1 could not separate the platform's own analytical capability from the contribution of a commercial reputation service consulted by millions of users, and because the malicious corpus is drawn from a source whose samples the antivirus industry has already catalogued. Experiment 2 uses 1,075 malicious packages spanning 15 named Android malware families.

Throughout, a sample is *caught* when the pipeline returns a malicious or suspicious verdict, and *missed* when it returns a benign verdict on a malicious file or fails to produce a verdict at all. Sandbox severity is reported on the scale used by the analysis platform, where 8 to 10 indicates malicious behaviour, 5 to 7 indicates suspicious behaviour, and below 5 indicates no malicious behaviour observed.

5.2 Experiment 1: Complete Pipeline

Of the 968 malicious samples, 52 could not be adjudicated because the VirusTotal API quota was exhausted before they were processed. These carry no reputation verdict, no static score and no sandbox result, and are excluded from the figures below rather than counted as successes, leaving 916 adjudicated samples.

Table 2 Confusion matrix and derived metrics for the complete pipeline over 916 adjudicated malicious and 1,116 benign samples.

	Predicted malicious	**Predicted benign**
Actually malicious (916)	907	9
Actually benign (1,116)	2	1,114
Derived metrics		
Detection rate (recall)		99.02%
Specificity		99.82%
False positive rate		0.179%
Precision		99.78%
F_1 score		99.40%
Balanced accuracy		99.42%

The false positive rate of 0.179% establishes that the deliberately conservative decision thresholds do not translate into a high false alarm rate. Table 3 shows how the benign corpus

distributed across the three static decision zones, and Figure 7 shows the underlying score distribution.

Table 3 Distribution of the benign corpus across the static decision zones. Escalated samples are not errors, but each represents a sandbox run the system would perform on harmless software.

Outcome	All (1,116)	AndroZoo (1,056)	F-Droid (60)
Cleared, score below 0.05	1,001 (89.70%)	943	58
Escalated, score 0.05 to 0.90	113 (10.13%)	111	2
False positive, score above 0.90	2 (0.18%)	2	0

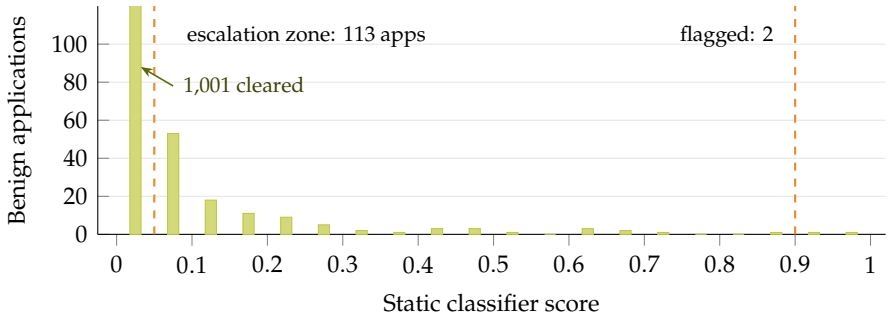


Figure 7 Static classifier scores over 1,116 benign applications. The vertical axis is clipped at 120 so the tail remains visible; the leftmost bar in fact contains 1,001 applications. The median score is 0.0058 and 700 applications score below 0.01. Only two cross the 0.90 boundary.

5.3 Experiment 2: Detection Without Reputation Data

With the VirusTotal lookup disabled, the platform relies solely on its own static analysis and sandboxed execution. Table 4 reports the outcome over 1,075 malicious samples.

Table 4 Outcome of the reputation-free deep scan over 1,075 malicious samples spanning 15 families, scored on the sandbox verdict alone. No VirusTotal query was made for any sample. Because the platform's severity-dominant integration rule (§4.3.3) would additionally treat a static malicious verdict as a catch, this figure is a lower bound on the reputation-free performance of the complete pipeline, which catches 1,056 of 1,075 (98.23%).

Outcome	Samples	Share
Malicious verdict (severity 8 to 10)	807	75.07%
Suspicious verdict (severity 5 to 7)	234	21.77%
Total caught	**1,041**	**96.84%**
Missed (severity below 5)	33	3.07%
Missed (execution error)	1	0.09%

The platform caught 96.84% of the corpus without consulting any external reputation source. Figure 8 places this alongside the Experiment 1 configurations and two reference points drawn

from the same reputation data: the VirusTotal aggregate verdict, and the average individual antivirus engine, computed as the mean proportion of engines returning a detection per sample.

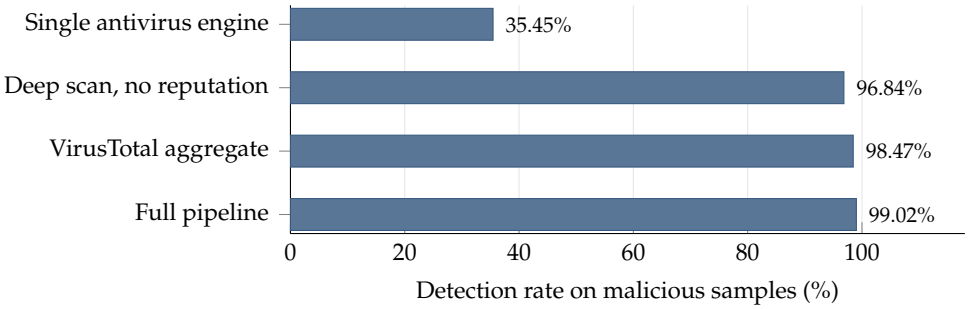


Figure 8 Detection rate by configuration. The first, third and fourth rows are measured on the 916 adjudicated samples of Experiment 1; the reputation-free deep scan (second row) is measured on the 1,075 samples of Experiment 2. The corpora differ, so the comparison is indicative rather than a controlled head-to-head.

5.4 Where the Detections Come From

Every sample in Experiment 2 carries the reason it was sent for sandboxed execution, which permits the sandbox stage to be evaluated against the static stage that preceded it. Table 5 and Figure 9 report the catch rate within each group.

Table 5 Sandbox catch rate by the reason the sample was escalated. The *model missed* row contains samples the static classifier scored as benign, and is the group that isolates the sandbox's independent contribution.

Escalation reason	Meaning	Samples	Caught	Rate
Model missed	Static classifier said benign	273	271	99.27%
Low score	Static score inside the middle zone	100	97	97.00%
Obfuscated	Static stage flagged packing or obfuscation	291	282	96.91%
Model detected	Static classifier already said malicious	328	313	95.43%
Previous run failed	Sandbox retry	81	77	95.06%
Scan error	Static stage errored	2	1	50.00%
All		**1,075**	**1,041**	**96.84%**

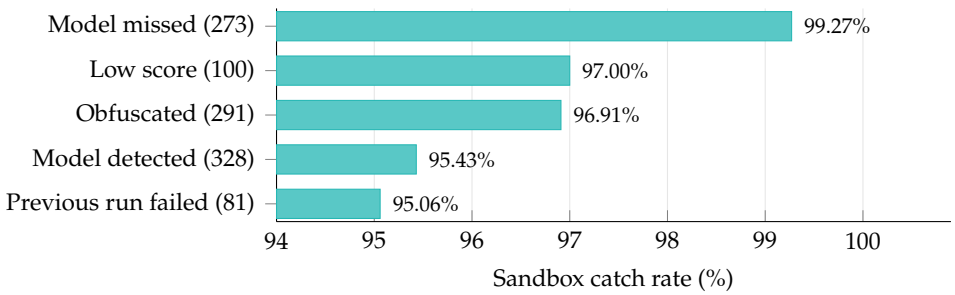


Figure 9 Sandbox catch rate by escalation reason. The horizontal axis begins at 94% so the differences between groups remain legible. The highest rate occurs precisely on the samples the static classifier had cleared.

5.5 Results by Malware Family

Table 6 reports Experiment 2 broken down by the 15 families present in the corpus.

Table 6 Reputation-free deep scan results by malware family. The obfuscation column gives the proportion of each family's samples that the static stage flagged as packed or obfuscated.

Family	Samples	Caught	Catch rate	Obfuscated
Joker	266	263	98.9%	32.7%
Cerberus	227	227	100.0%	24.2%
Hydra	153	129	84.3%	45.8%
Coper	123	120	97.6%	0.0%
Alien	92	91	98.9%	45.7%
Medusa	32	32	100.0%	0.0%
BRATA	31	31	100.0%	3.2%
Anubis	29	27	93.1%	13.8%
FluBot	27	27	100.0%	51.9%
SharkBot	26	26	100.0%	0.0%
Ermac	26	26	100.0%	11.5%
Hook	25	24	96.0%	24.0%
TeaBot	13	13	100.0%	61.5%
Harly	4	4	100.0%	0.0%
BlackRock	1	1	100.0%	100.0%
All	**1,075**	**1,041**	**96.84%**	**27.1%**

Nine of the fifteen families were detected without a single miss. The 34 misses are heavily concentrated: 24 of them belong to Hydra, which alone accounts for 70.6% of all failures despite representing 14.2% of the corpus. Excluding Hydra, the catch rate across the remaining 922 samples is 98.92%. The sandbox additionally attributed a specific malware family to 721 of the 1,075 samples, 67.1% of the corpus, and every one of those attributions occurred on a sample that

was caught.

5.6 Scope of the Evaluation

- **Different corpora.** Experiments 1 and 2 use overlapping but distinct sample sets, so the detection rates in Figure 8 are indicative rather than a controlled comparison on identical inputs.
- **No benign corpus in Experiment 2.** The reputation-free configuration was measured on malware only, so its false positive rate is unmeasured. The 0.179% figure comes from Experiment 1 and applies to the complete pipeline.
- **Quota exhaustion.** 52 of 968 malicious samples in Experiment 1, 5.37% of that corpus, were never adjudicated because the VirusTotal quota ran out.
- **Corpus provenance.** Both malicious corpora come from AndroZoo, so the samples are files the antivirus industry has already catalogued. This inflates the apparent strength of the reputation stage in Experiment 1, and makes Experiment 2 the more representative measurement of the platform's own capability on unfamiliar software.
- **Benign label quality.** AndroZoo benign labels derive from detection counts at collection time rather than manual review, so a mislabelled sample would appear as a false positive.

6 Discussion

6.1 The Platform Detects Malware Without Reputation Data

The most important result in this evaluation is that removing VirusTotal entirely costs the system relatively little. With no reputation lookup of any kind, the platform caught 96.84% of 1,075 malware samples, against 98.47% for the VirusTotal aggregate verdict measured on the comparable corpus of Experiment 1.

The significance of that gap lies in what sits on either side of it. VirusTotal aggregates roughly 74 commercial antivirus engines, is operated at global scale and is consulted by millions of users, and its verdicts on these samples reflect years of accumulated industry analysis of files it has already seen. On the other side is a student-built pipeline consisting of one gradient-boosted classifier trained on a public dataset and one sandbox run per sample. That this pipeline lands within 1.6 percentage points of the aggregate, using none of its data, is the strongest evidence available that the analysis is doing real work rather than relaying someone else's conclusions.

The comparison also reverses the concern raised by Experiment 1. There, the reputation stage resolved 902 of 907 catches, and the platform's marginal contribution appeared to be 0.55 percentage points, which invited the reading that VirusTotal was doing all the work. Experiment 2 shows that this reading was an artefact of stage ordering rather than of capability. Because the reputation lookup runs first and is cheap, it resolves most samples before the later stages are ever invoked, and a stage that is rarely invoked will always appear to contribute little. When it is invoked on everything, it recovers almost the same set of files independently.

This distinction matters for the deployment the system targets. Sideloaded and third-party packages are precisely the population where reputation coverage is weakest, and a file no engine has ever submitted returns *unknown* rather than a verdict. On those files the reputation stage contributes nothing at all, and the 96.84% figure, not the 0.55 point margin, is the relevant estimate of what the user actually receives.

6.2 The Sandbox Recovers What the Classifier Misses

The escalation-reason breakdown provides the cleanest available evidence for the layered design, because it isolates each stage's behaviour on samples the other stage got wrong.

The highest catch rate in Table 5, 99.27%, occurs on the 273 samples the static classifier had scored as benign. On files where the classifier was not merely uncertain but confidently incorrect, sandboxed execution recovered 271 of 273. The rate on samples the classifier had already flagged as malicious was lower, at 95.43%. The samples the static model most needed help with are precisely those on which the sandbox performed best.

This is the complementarity the architecture assumes, demonstrated rather than asserted. It also indicates that the two stages fail on different files rather than on the same hard subset, which is the property that makes layering worthwhile at all. Had the sandbox failed on the same samples the classifier failed on, the additional cost of execution would have bought nothing.

6.3 Obfuscation Is Not What Causes Misses

An intuitive reading of Table 6 is that obfuscation drives the failures, since packing and runtime unpacking are exactly what static analysis cannot see through. The data does not support this.

Samples escalated specifically because the static stage flagged them as obfuscated were caught at 96.91%. Every other sample was caught at 96.81%. The difference is one tenth of a percentage point. Across families the relationship is equally absent: TeaBot, at 61.5% obfuscated, was caught without a single miss, as were FluBot at 51.9% and BlackRock at 100%, while Coper, Medusa and SharkBot recorded no obfuscation at all and Coper still produced misses in the first of those three.

The explanation is that obfuscation is a defence against static inspection, and the stage being measured here is not static inspection. Packing conceals code from a decompiler, but the packed application must still unpack and execute its payload to accomplish anything, and at that point the behaviour is visible to instrumentation regardless of how well the bytecode was hidden. This is the mechanism the obfuscation escalation policy was designed to exploit, and the result confirms that the policy works: routing obfuscated files to behavioural analysis neutralises the evasion rather than merely flagging it.

6.4 Hydra Is a Family-Specific Failure

If obfuscation does not explain the misses, something else must, and the distribution points to a single family. Hydra contributes 24 of the 34 failures while making up 14.2% of the corpus, and removing it lifts the catch rate from 96.84% to 98.92%.

Two details suggest the cause is behavioural rather than structural. First, Hydra's misses are not marginal: all 24 scored 1 or 3 on the severity scale, meaning the sandbox observed essentially nothing during execution, rather than observing something ambiguous. Second, the misses are spread across escalation reasons, 12 from samples the classifier had detected, 7 obfuscated, 4 retries and 1 low score, so they are not an artefact of any single routing path. Hydra's obfuscation rate of 45.8% is also almost identical to Alien's 45.7%, and Alien was caught at 98.9%.

The most plausible reading is that these Hydra samples did not execute their payload during the observation window, whether through dormancy, an unreachable command server, or environment checks that suppressed activity. All three are known limitations of time-bounded dynamic analysis rather than defects in this implementation, and all three would be addressed by the same intervention: longer execution windows and simulated user interaction. Because the failures are concentrated in one family, this is a targeted and tractable next step rather than a general weakness.

6.5 Detection Versus Decisiveness

A caveat applies to the 96.84% figure that the headline number conceals. Of the samples caught, 807 received a malicious verdict but 234 received only a suspicious one. On a safety reading the two are equivalent, since neither tells the user a file is safe. In use they are not equivalent, because a suspicious verdict returns the decision to the user, which is the burden the system exists to remove.

The complete pipeline of Experiment 1 does noticeably better here, since a conclusive reputation verdict converts many of those cases into definitive answers. The reputation stage therefore contributes less to whether malware is caught than to how confidently the result can be stated. That is a real contribution, and it is a more accurate description of its role than the raw detection figures in Figure 8 suggest.

6.6 Where the Results Meet and Miss Expectations

The reputation-free result exceeds expectations. The system was designed on the assumption that its local stages would serve as a safety net beneath a stronger reputation signal, and the finding that those stages independently approach the aggregate performance of the reputation service was not anticipated.

The false positive rate meets expectations, at 0.179% with zero false positives on the curated F-Droid subset, and confirms that conservative thresholds did not produce an unusable alarm rate.

Two gaps remain. Experiment 2 was run on malware only, so the reputation-free configuration's false positive rate is unmeasured, and it cannot be assumed that a pipeline relying more heavily on its own classifier would maintain the 0.179% figure. Repeating Experiment 2 against the benign corpus is the immediate next step. Separately, both malicious corpora are drawn from AndroZoo and are therefore already catalogued by the antivirus industry. This biases Experiment 1 in favour of the reputation baseline, and it means the 96.84% figure, while measured without reputation data, was still measured on samples that are not genuinely novel. A corpus of recently discovered malware would test the claim that matters most for the deployment the system targets.

7 Limitations and Future Work

7.1 Sandbox Evasion

Malware that detects an analysis environment may suppress its behaviour and produce a clean behavioural report. This is a limitation of dynamic analysis as a technique rather than of this implementation. A consequence worth stating plainly is that a benign behavioural verdict is weaker evidence than a malicious one: observed malicious behaviour is conclusive, whereas observed inactivity is ambiguous between genuine benignity and successful evasion.

7.2 External Dependencies

Analysis depends on three external services: VirusTotal, Tria.ge, and Gemini. Availability, rate limits, and interface changes at any of them affect the system.

7.3 Reduce External Dependency

Supporting a locally-hosted language model for report generation would remove the requirement that findings be transmitted to a third party, enabling use in environments where confidentiality precludes external submission.

8 Conclusion

This project set out to close two gaps in Android malware analysis: an accessibility gap, in which analyst-grade findings are unreadable to the users who most need protection, and an operational gap, in which existing tools run every analysis stage on every file regardless of need. The resulting platform addresses both through a cost-aware tiered pipeline and an explanation layer that preserves auditability. Analysis proceeds through a local hash cache, a VirusTotal reputation lookup, static classification, and behavioural sandbox execution, in ascending order of cost, with each tier filtering the next. Because the escalation conditions compound, the most expensive stage is reached only by files that are simultaneously unknown to the cache, unknown to VirusTotal, statically inconclusive, and submitted under Deep Scan. Verdicts and scores are derived entirely from reputation data, model inference and sandbox observation, and the language model is confined to producing prose, so that every decision remains deterministic and attributable to specific evidence while a single generation call still yields two audience-specific reports, one in plain language and one for practitioners.

The evaluation supports both claims quantitatively. The complete pipeline detected 99.02% of adjudicated malware while misclassifying 2 of 1,116 benign applications, a false positive rate of 0.179%, with no false positives at all on the curated F-Droid subset. More importantly, with the VirusTotal lookup disabled entirely, the platform still caught 96.84% of 1,075 malware samples spanning 15 families using only its own static classifier and sandbox. That figure sits within 1.6 percentage points of the VirusTotal aggregate verdict, which combines roughly 74 commercial engines and is consulted by millions of users, and it is close to three times the 35.45% achieved by an average individual engine on a comparable corpus. The layered design is also justified by where the stages disagree rather than by their totals: on the 273 samples the static classifier confidently cleared, sandboxed execution recovered 271, a rate of 99.27% and the highest of any group. The two stages fail on different files, which is the property that makes paying for execution worthwhile.

The main takeaway is that accessible malware analysis need not be shallow analysis. A pipeline that decides how much scrutiny each file warrants, rather than applying the same expensive treatment to everything, achieved detection comparable to a global commercial reputation service without relying on it, and returned that verdict as an explanation a non-technical user can act on. The remaining weaknesses are specific and tractable rather than structural: 24 of the 34 total misses belong to a single family whose samples stayed dormant during execution, which points to longer observation windows and simulated user interaction as the next concrete step, and the reputation-free configuration still needs to be measured against the benign corpus before its false positive rate can be claimed.

References

[1] Connor Tumbleson and Ryszard Wiśniewski. *Apktool: A tool for reverse engineering Android APK files*. https://apktool.org. Accessed: 2026-07-09. 2026.

[2] MobSF Team. *Mobile Security Framework (MobSF)*. https://github.com/MobSF/Mobile-Security-Framework-MobSF. Accessed: 2026-07-09. 2026.

[3] VirusTotal. *VirusTotal*. https://www.virustotal.com. Accessed: 2026-07-09. 2026.

[4] zinja-coder. *apktool-mcp-server*. https://github.com/zinja-coder/apktool-mcp-server. Accessed: 2026-07-09. 2026.

[5] Google. *Google Play Protect*. https://developers.google.com/android/play-protect. Accessed: 2026-07-09. 2026.

[6] Koodous. *Koodous: Android Malware Analysis Platform*. https://koodous.com. Accessed: 2026-07-09. 2026.

[7] AndroZoo. *AndroZoo: Collecting Millions of Android Apps for the Research Community*. https://dl.acm.org/doi/10.1145/2901739.2903508. DOI: 10.1145/2901739.2903508. Accessed: 2026-08-08. 2016.