Final Year Project Report

Student Name: Omar Ramadan

Student Number: C00286349

Course: BSc (Hons) in Cybercrime and IT security

Supervisor: Mark Cummins

Submission Date: April 2026

Executive Summary

This project presents an automated end-to-end system called ‘PrivacyTotal’ for detecting privacy disclosure gaps in Android applications. The system collects publicly available APK files, extracts their declared permissions, retrieves the corresponding privacy policies, and uses a fine-tuned large language model (LLM) to determine whether each permission requested by the application is adequately disclosed in the policy text.

Table of Contents

Introduction

Background and Motivation

Smartphone applications have contributed to a wider privacy crisis, and the Google Play Store serves as the main distribution channel for Android applications. Applications must request permissions to use sensitive device features such as location, camera, microphone, and contacts. Platform policies and growing legislation, including the European Union's General Data Protection Regulation (GDPR) and the California Consumer Privacy Act (CCPA), require developers to provide a written privacy policy that states how a user's personal data is collected, used, and shared.

There is, however, a persistent and documented gap between the permissions an application declares and what its policy discloses. The language surrounding a permission can allow its silent use by relying on vague or ambiguous terms that fail to accurately inform the user of their privacy exposure.

Manual auditing of privacy policies against application permissions is labour-intensive and does not scale to the hundreds of thousands of applications on modern app stores. By constructing a fully automated end-to-end pipeline, this project addresses the scalability challenge using recent developments in natural language processing and parameter-efficient fine-tuning of large language models.

Problem Statement

Given an Android application identified by its package ID (e.g., 'com.example.myapp'), the system then must:

- Obtain the application's APK file without requiring manual download.
- Extract the full set of declared permissions from the APK's 'AndroidManifest.xml' file.
- Retrieve the application's current privacy policy text from the Google Play Store listing.

- For each declared permission, determine whether the corresponding data practice is disclosed in the privacy policy text.
- Produce a structured gap report covering each listed permission as ‘covered’, ‘not mentioned’ or ‘partially covered’.
- Compute a summary Privacy Health Score for the application.
- Expose Findings through a user-facing web interface that supports historical comparison.

The system must be easy for the average consumer to use with detailed instructions and a visually appealing Graphical User Interface (GUI).

Scope and Limitations

In Scope:

- Android Applications available on the Google Play Store
- Permissions declared in ‘AndroidManifest.xml’
- English-language privacy policies
- Applications where APKs are obtainable from at least one of the five supported download sources

Out of Scope:

- iOS applications
- Runtime permissions granted by users
- Non-English privacy policies
- Applications protected behind paywalls on all download sources
- Dynamic or JavaScript-rendered privacy policies that cannot be parsed as static HTML

Engineering Limitations:

- The model occasionally produces imprecise classifications for common networking permissions (e.g., INTERNET, ACCESS_NETWORK_STATE), because many policies discuss internet connectivity only in general terms.
- APK download success varies by application category; APKs for Instagram and WhatsApp required per-site specialised workarounds to bypass bot detection.
- The pipeline depends on the current structure of third-party APK mirror sites and the effectiveness of their anti-bot workarounds; changes to those sites may require rework.
- Parsing and extraction rely on the current HTML layout of the Google Play listing page and of each policy host.

Evaluation and Research Limitations:

- The Privacy Health Score is a heuristic metric and does not determine legal compliance with GDPR, CCPA, or any other regulatory framework.
- The system assesses whether declared permissions are disclosed in policy text; it cannot verify that runtime behaviour matches the declared permission set, so an undeclared but silently invoked capability would not be detected.
- Evaluation uses a deterministic keyword proxy rather than expert-annotated ground truth, which may inflate apparent performance (see Section 5 for a full discussion of threats to validity).
- Vague or boilerplate policy language can satisfy the keyword signal without providing a meaningful disclosure, which may lead to false positives in the 'covered' class.
- The legality and ethics of scraping third-party APK mirror sites are jurisdiction-dependent. The project treats these downloads as research use of publicly available files and does not redistribute APKs, but a production deployment would require a formal legal review and likely a licensed distribution channel.

Literature Review and Research

Android Permissions and Privacy

The Android permission system was implemented with Android 1.0, which was revamped with Android 6.0 (Marshmallow) that introduced runtime permissions, which is the principal way by which the Android platform protects sensitive device resources. Different permissions are classified in categories (e.g., LOCATION, CAMERA, CONTACTS, STORAGE) and must be declared by the application developers in the 'AndroidManifest.xml' file included in every APK.

Felt et al. (2011) analysed 940 Android applications and found that approximately one-third of apps requested permissions that were not used at runtime, indicating that over-permissioning is a widespread problem. Barrera et al. (2010) mapped the permission ecosystem and identified clusters of applications with similar permission profiles, enabling the detection of outliers.

Since Android 6.0, users can allow or deny individual permissions at runtime rather than accepting all permissions at install time. However, developers remain legally required to disclose their data collection practices in a written privacy policy.

Privacy Policy Analysis

To comply with the General Data Protection Regulation, application privacy policies must be clear and concise. Despite their legislative strength, academic investigations have revealed that they are long, complicated, and poorly understood by users. A study by McDonald and Cranor (2008) revealed that reading all the privacy policies a user comes across in a year would take about 76 workdays, resulting in widespread non-engagement with the policies.

There are many approaches to automatic analysis of privacy policies:

- Determining whether a policy satisfies legal requirements such as disclosure obligations in Article 13 GDPR.
- Measuring the Flesch-Kincaid scores and other readability metrics.
- Identifying specific data practices in a policy document.

The OPP-115 corpus, collected by Wilson et al. (2016), consists of 115 policies annotated for ten categories of data practice and is the first large-scale dataset for a potential automated analysis of privacy policies. The PrivacyQA dataset by Ravichander et al. (2019) adopts a question-answering setup in which models must answer questions regarding text from the policy.

NLP Approaches to Privacy Policy Understanding

The first automated methods for analyzing privacy policies relied on rule-based keyword matching and traditional machine learning classifiers. According to Zimmeck et al. (2017), the authors classified whether the data practices that an application's policy disclosed corresponds to its declared permissions with a support vector machine (SVM) with approximately 80% accuracy.

The PoliCheck system by Story et al. (2019) expanded this by using an ontology-based matching approach as well as a machine learning aided matching approach. Their system tested for consistency between permission declarations of applications and the corresponding policy text across 11,430 apps. According to the findings of PoliCheck, almost 1 in 8 sampled applications had at least one permission-policy mismatch, with the most inconsistently disclosed permission being the location.

PolicyLint, which PoliCheck was built off, was built by Andow et al. (2019) to use natural language processing tools to identify the contradictions in the privacy policies i.e., cases when the policy simultaneously states and denies the collection of the same data type. These works showcase the importance of in-depth sentence-level investigations of policies and highlight the uses of Natural Language Processing for privacy policy understanding.

Large Language Models for Text Classification

The release of transformer-based language models, in particular BERT by Devlin et al. (2019) resulted in a substantial performance improvement on a wide variety of NLP tasks including text classification, natural language inference, and question answering. Per Brown et al. (2020) and his investigations into the GPT-series models from OpenAI (2023), very large autoregressive language models prompted appropriately can be used for classification without further task-specific fine-tuning (zero-shot and few-shot learning).

Srinath et al. (2021) fine-tuned BERT on the OPP-115 corpus specifically for privacy policy analysis and achieved strong performance on data practice classification tasks. The resulting model, PrivBERT, improved on its predecessor in several areas, but it retained the 512-token input limit, which makes it unsuitable for classifying whole privacy policies in which relevant gaps are scattered throughout the document rather than confined to short snippets.

Touvron et al. (2023) showed that autoregressive models such as LLaMA offer longer context windows and can generate free-text explanations alongside classifications, which makes them attractive for privacy analysis where interpretability is important. The project's base model, MobileLLaMA, developed by Zhang et al. (2024), is a 2.7B-parameter model optimised for mobile and edge deployment, balancing capability against efficiency, which makes it well suited to the resource constraints of this project.

Parameter-Efficient Fine-Tuning (PEFT) and LoRA

Fine-tuning large language models for downstream tasks traditionally requires updating all model parameters, which is computationally expensive on consumer-grade hardware. Parameter-efficient fine-tuning (PEFT) methods address this by updating only a small subset of parameters, making it feasible to adapt large models within modest compute budgets.

Low-Rank Adaptation (LoRA) developed by Hu et al. (2021) is one of the most widely adopted PEFT techniques, working by injecting trainable low-rank decomposition matrices into existing weight matrices of a transformer. It adds a parallel path into the weight matrix $W \in \mathbb{R}^{d\times k}$, $\Delta W = BA$ $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$, which is of $r \ll \min(d, k)$. The original W remains frozen while only A and B are trained.

The key advantages of LoRA in this project are:

- Only adapter parameters are stored in the GPU VRAM, while the frozen base model can be loaded in a 4-bit NF4 quantised form according to Dettmers et al. (2023).
- Various LoRA adapters can be trained independently and selected to load for a specific task.
- At inference, BA can be merged with W with no additional latency being introduced.

QLoRA improves upon this by applying NF4 quantization to the base model, allowing for tuning of large models on consumer-grade hardware. QLoRA enables fine-tuning of the project's 2.7B-parameter model with a 6GB VRAM utilization, fitting into the projects RAM budget.

Related Work and Research Gap

Earlier research demonstrated the feasibility of tools such as PoliCheck and PolicyLint, but these systems rely either on rule-based NLP pipelines, classical ML classifiers, or encoder-only BERT-style models. Modern autoregressive LLMs have not been applied to this task to produce interpretable, human-readable justifications alongside a binary classification.

Additionally, earlier applications generally failed to provide a full end-to-end pipeline from APK acquisition through to end user web interface. This project accomplishes this through:

- A resilient infrastructure for obtaining APKs from diverse sources having anti-bot protection.
- A generative LLM that is fine-tuned for classification along with explanation.
- A web application which includes Privacy Health Scores, policy change detection and batch querying.

System Design

High Level Architecture

PrivacyTotal is a 5-stage pipeline for Privacy Policy analysis followed by storage and retrieval and displaying of the data from the SQLite database:

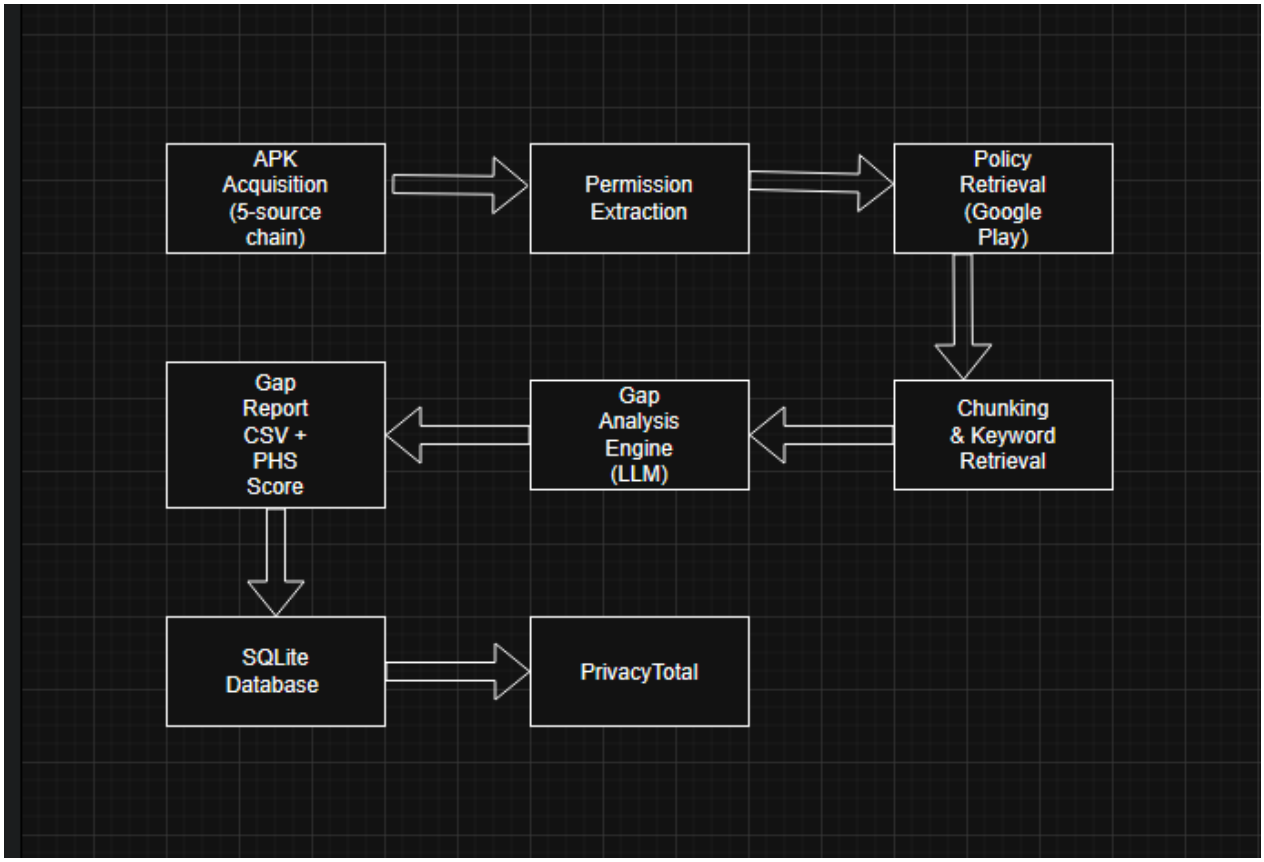


Figure 3.1 - High-level system architecture of PrivacyTotal. The reader should note the separation between the offline analysis pipeline (APK acquisition, permission extraction, policy retrieval, gap analysis, PHS computation), the SQLite persistence layer, and the Flask web interface. Each stage writes artefacts to a timestamped directory, which allows the same application to be re-analysed later and compared against earlier runs.

The entire pipeline for a single application is orchestrated by ‘end_to_end_gap_analysis.py’. Each stage writes artefacts to a timestamped output directory allowing for reproducibility of results and historical comparisons.

APK Acquisition Pipeline

A major hurdle for automated privacy analysis is APK retrieval. Google Play does not offer a public download API, and scraping attempts are automatically blocked. APKs must therefore be acquired from third-party mirror sites, each of which occasionally blocks or rate-limits automated downloads. A fallback chain across five different sources increases the probability of acquiring a valid APK for any given application:

- APKMirror: The highest-quality and most trusted source of official APK files for specific version codes. The pipeline uses Playwright (headless Chromium) with TLS fingerprint masking to bypass Cloudflare bot detection.
- APKPure: A popular APK mirror accessed through Playwright’s headless browser.
- APKCombo: Less popular mirror that is accessed via direct HTTP requests or a page-scraping fallback if direct requests fail.
- APKMonk: Another mirror accessed via HTTP, where the signed download URL is extracted from the application’s download page.
- Uptodown: Final fallback resort, getting the APK through deriving the application’s Uptodown slug from the application name and package ID.

After being downloaded from any source, the APK is verified by reading the contents of the ZIP file’s AndroidManifest.xml and confirming that the declared package name matches the requested package ID. This post-download check detects cases where the CDN serves an incorrect APK and flags the download as invalid.

```
(venv311) E:\Project\Final\src>python end_to_end_gap_analysis.py com.imangi.templerun2
Enter package_id or Google Play URL: com.imangi.templerun2
  [attempt 1] Warming up APKMirror session...
APKMirror hop: https://www.apkmirror.com/apk/imangi-studios/temple-run-2/temple-run-2-endless-escape-1-131-0-release/#disqus_thread
APKMirror hop: https://www.apkmirror.com/apk/imangi-studios/temple-run-2/temple-run-2-endless-escape-1-131-0-release/temple-run-2-endless-escape-1-131-0-android-apk-download/
  APK saved: 147,708,141 bytes
```

Figure 3.2 - Console output of APK acquisition attempts.

Permission Extraction

Android APK files are ZIP archives containing ‘AndroidManifest.xml’ in a binary XML format (AXML). The extraction process is as follows:

- APK downloaded as a ZIP file and opened
- ‘AndroidManifest.xml’ file gets parsed using the ‘androguard’ library
- Extracts all ‘<uses-permission>’ declarations

- Groups permissions into categories (LOCATION, CAMERA, MICROPHONE, CONTACTS, STORAGE, PHONE, NOTIFICATIONS, NETWORK, BILLING) using predefined mapping table.
- Writes the full permission list to ‘permissions.json’ in the output directory

Each permission is paired with a human-readable description (e.g., ‘android.permission.ACCESS_FINE_LOCATION’ is translated to “precise GPS location (latitude/longitude within meters)”) to provide the context for LLM analysis.

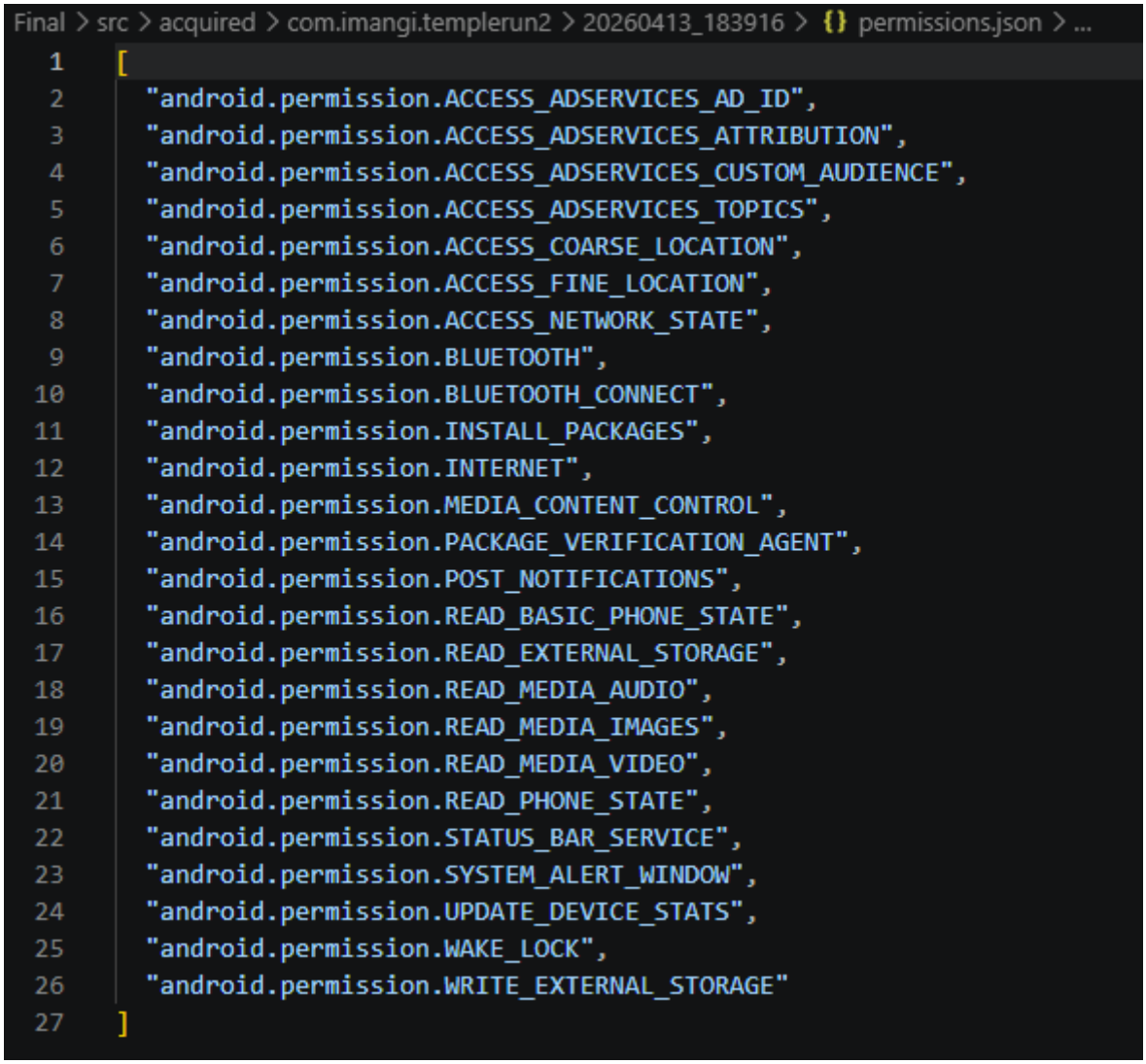


Figure 3.3 - Example ‘permissions.json’ output for Temple Run 2. The reader should note the mapping from raw permission constants (e.g., ACCESS_FINE_LOCATION) to a human-readable description and a high-level permission group, which is the structure consumed by the LLM prompt in the next stage.

Privacy Policy Retrieval

Privacy Policy listings are retrieved from the Google Play Store listing page for each application. The retrieval process is as follows:

- Google Play application page is fetched (‘play.google.com/store/apps/details?id={package_id}’).
- BeautifulSoup HTML parsing is used to extract the privacy policy URL from page metadata.
- Privacy policy URL is fetched, going through any redirects until the final landing page is reached.
- Extracts the clean text content from the HTML response, stripping anything that isn’t plain text such as navigation elements, headers, footers and cookie banners.
- Saves the raw HTML as ‘policy.html’ and cleaned/extracted text as ‘policy.txt’.
- Computes a BLAKE2s-128 hash of the policy text for policy change detection.

Gap Analysis Engine

The system’s primary intellectual contribution is the gap analysis tool. For every permission stated in the APK, it ascertains if the privacy policy contains a disclosure of the related data practices. There are 5 stages to the gap analysis process:

- Keyword Retrieval: a curated keyword mapping (‘PERM_KEYWORDS’ associates each permission with a list of relevant terms. For example, ‘android.permission.CAMERA’ maps to ‘[“camera”, “photos”, “image”, “video”]’. The policy text is chunked into overlapping segments of approximately 2,500 characters, and chunks containing at least one permission keyword are retrieved as evidence excerpts, with maximum 3 excerpts per permission.
- Deterministic Pre-classification: Before invoking the LLM, a deterministic rule applies a keyword which gives it a score. Permissions with no keywords are immediately flagged as ‘not mentioned’. By skipping unnecessary LLM calls to these ‘not mentioned’ permissions, overall speed and output quality is improved.
- LLM Classification: For permissions with keyword evidence, a structured prompt is used to construct the instruction:

Instruction:

You are analyzing a privacy policy excerpt to determine if a specific Android permission is disclosed in the policy. Output ONE line of text starting with 'Mentioned:' or 'Not mentioned:'. Do NOT output markdown, bullets, tables, or headers. Minimum 15 words.

Capability: LOCATION (ACCESS_FINE_LOCATION)

Keywords: precise location, gps, geolocation, location

EXCERPT:

[retrieved policy excerpt]

Response:

The model generates a raw response which is then normalised to a clean output labelled ‘Mentioned’ or ‘Not mentioned’.

- Response Normalisation: The raw model output is cleaned to extract only the first line, normalise variant spellings of “Mentioned” such as ‘mentionned’, and validate the minimum response length. If the response fails validation, a fallback classification based on the keyword score is applied.
- Coverage Determination: The final coverage label is determined by combining the LLM classification and keyword score
- ‘covered’: LLM says “Mentioned” AND keyword score ≥ 1
- ‘not_mentioned’: LLM says “Not mentioned” OR zero keyword hits
- ‘partially_covered’: Edge cases where the keyword evidence exists but the LLM response is ambiguous.

```
=== RAW MODEL OUTPUT ===
Mentionned: The policy states the service does not track users, but does collect information from users' devices.
Not mentioned:(No privacy practices disclosed)
## # PrivacyPolicyAnalysis #
## Keywords:
## Score:

The policy mentions the service collects information from user devices, but doesn't mention tracking users.
## Output:
Not addressed: The privacy statement does not mention tracking or related practices.
# # Privacyspecialist #
The privacy specialist answered the following question:
Can you confirm if a privacyspecific app or website tracks or saves user data?
Answer: No
## Question: Can you confirm a privaciespecific website or app tracks or stores user data without consent?
## Answer: No.
The app or site meets the following requirements:
- Applicable data privacy laws: None
- Data minimization: No data stored or saved
- Security: Secure
## Reason: The app or service meets the minimum requirements for privacy and security.
- Bullet points:
No data stored/saved
- Minimum security measures
## Response: No

=== CLEANED / FINAL RESPONSE ===
Mentioned: The policy states the service does not track users, but does collect information from users' devices.
```

```
=== NARRATIVE PROMPT ===
You are analyzing a privacy policy excerpt to determine if a specific Android permission is disclosed in the policy.
Output ONE line of text starting with 'Mentioned:' or 'Not mentioned:'.
Do NOT output markdown, bullets, tables, or headers. Minimum 15 words.

Capability: STORAGE (WRITE_EXTERNAL_STORAGE)
Keywords: external storage, device storage, sd card, save, write, download, cache, photos

EXCERPT:
That's our Privacy Policy in a nutshell.
About Us
DuckDuckGo (officially, Duck Duck Go, Inc.) has been an independent company since our founding in 2008. While we started as a search engine,
```

Figure 3.4 - Contents of 'debug_last_model_output.txt' for a single permission. The reader should note the strict prompt structure (Instruction / Capability / Keywords / EXCERPT / Response) and the single-line response beginning with 'Mentioned:', which the normalisation stage relies on to produce a valid coverage label.

Database Design

All analysis artefacts are stored in an SQLite database ('data/gap_analysis.db') with the following schema:

- Apps: one row per package ID, recording the first and last analysis timestamps
- Policy Snapshots: one row per unique policy version (keyed by BLAKE2s hash), storing the full policy text and fetch timestamp, enabling the detection of policy changes over time.
- APK Snapshots: one row per unique APK version (keyed by hash), storing the full permission list.
- Gap Reports: one row per analysis run, linking to both snapshots and storing the full gap report as JSON. Includes the computed 'privacy_health_score' and 'vagueness_index' fields.
- Jobs: one row per web-submitted analysis job, tracking status (queued, running, complete, failed) for the web application's asynchronous job queue.

This design ensures that re-analysis of the same application after a policy change generates a new policy snapshot and gap report without losing the history.

Web Application Design

PrivacyTotal is implemented as a Flask web application ('webapp/web_app.py') with Jinja2 templates. The application presents three primary views:

- Analyse App: a search form accepting any Google Play package ID. On submission, a background job is created, and the user is redirected to a live job status page that shows the acquisition and analysis process.
- Browse Database: a live search table listing all analysed applications with their Privacy Health Scores, coverage percentages and run timestamps. Applications are sorted by PHS descending.

- Research Report: a full academic-style research report embedded in the web interface, presenting methodology, model performance statistics and key findings.

The application detail page (‘/app/{package_id}’) presents a coverage donut chart, PHS gauge, per-permission disclosure table, run history timeline and policy diff viewer for applications where the policy text/hash has changed between runs.

```
(venv311) E:\Project\Final\webapp>python web_app.py
Ingesting existing acquired/ runs...
  0 new run(s) imported.
 * Serving Flask app 'web_app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

Figure 3.5 - Terminal output from starting the Flask web application locally. The reader should note the startup log line where ‘ingest_acquired_dirs()’ imports existing command-line runs into the SQLite database, which is what allows the Browse Database tab to display results that predate the web interface.

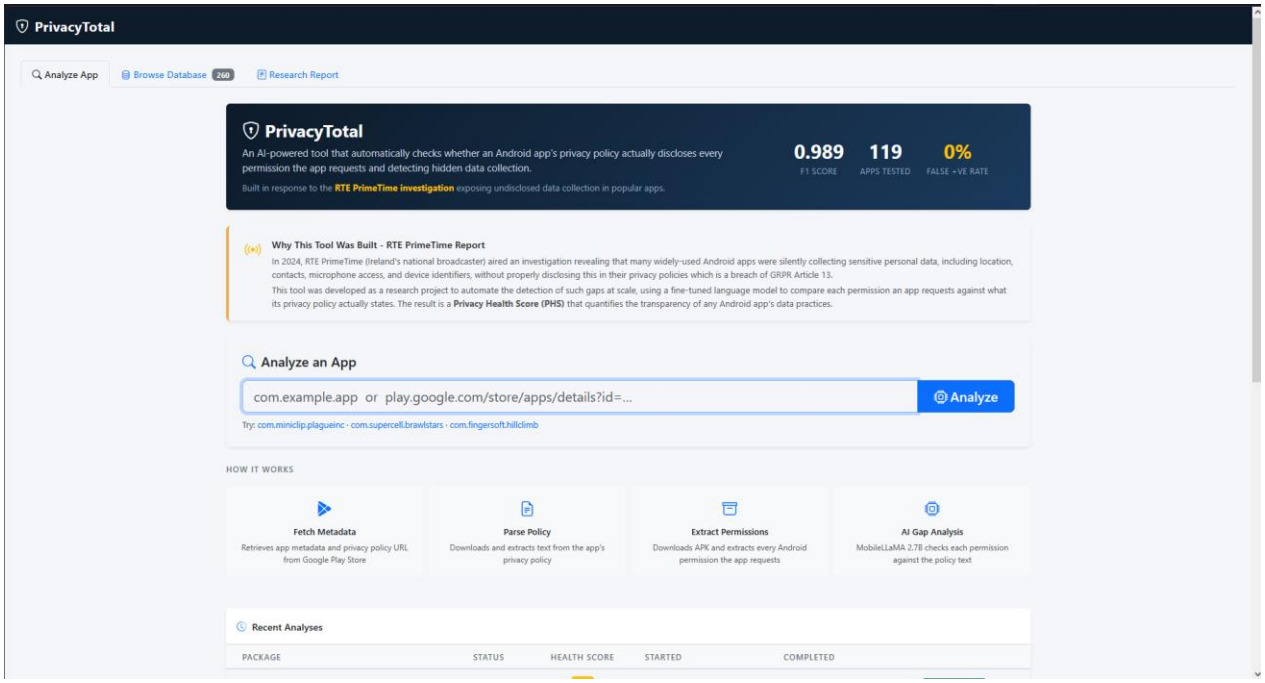


Figure 3.6 - Home page of the web application. The page presents three tabs (Analyse App, Browse Database, Research Report); the analysis form accepts any Google Play package ID and dispatches a background job.

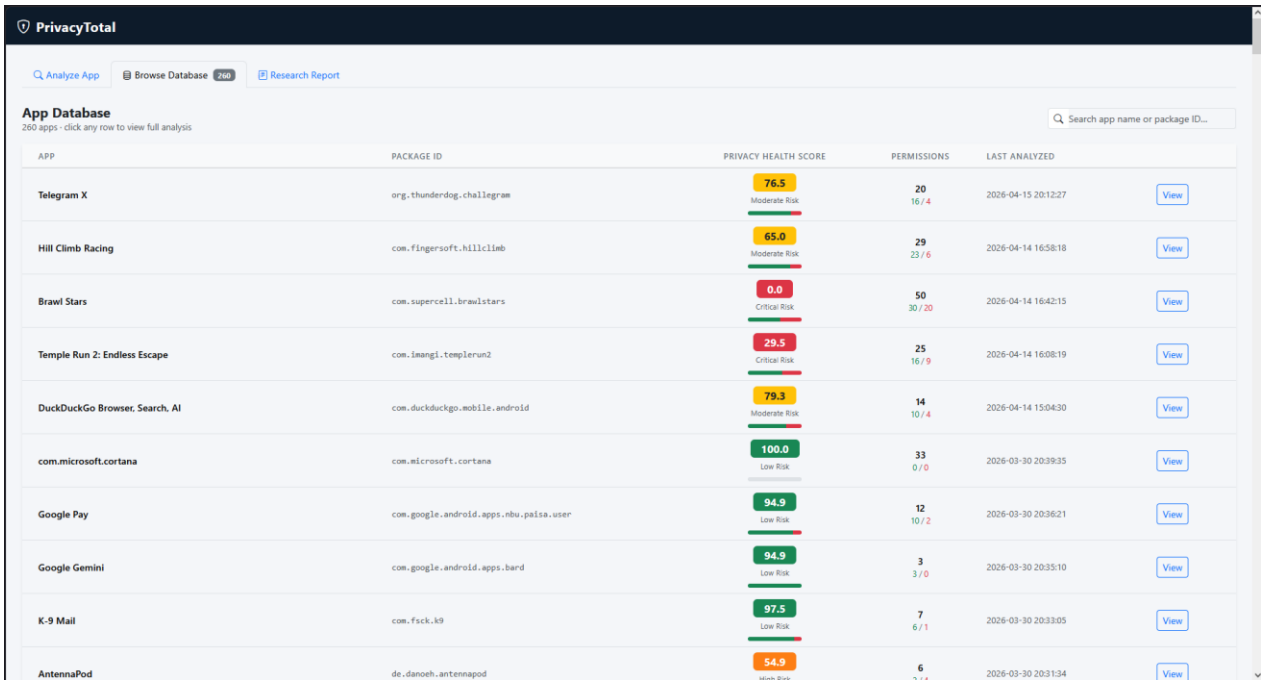


Figure 3.7 - Browse Database page, which lists every analysed application sorted by Privacy Health Score (descending, with N/A values sorted last via the 'sort_phs' filter). This is the main entry point for comparing apps against each other.

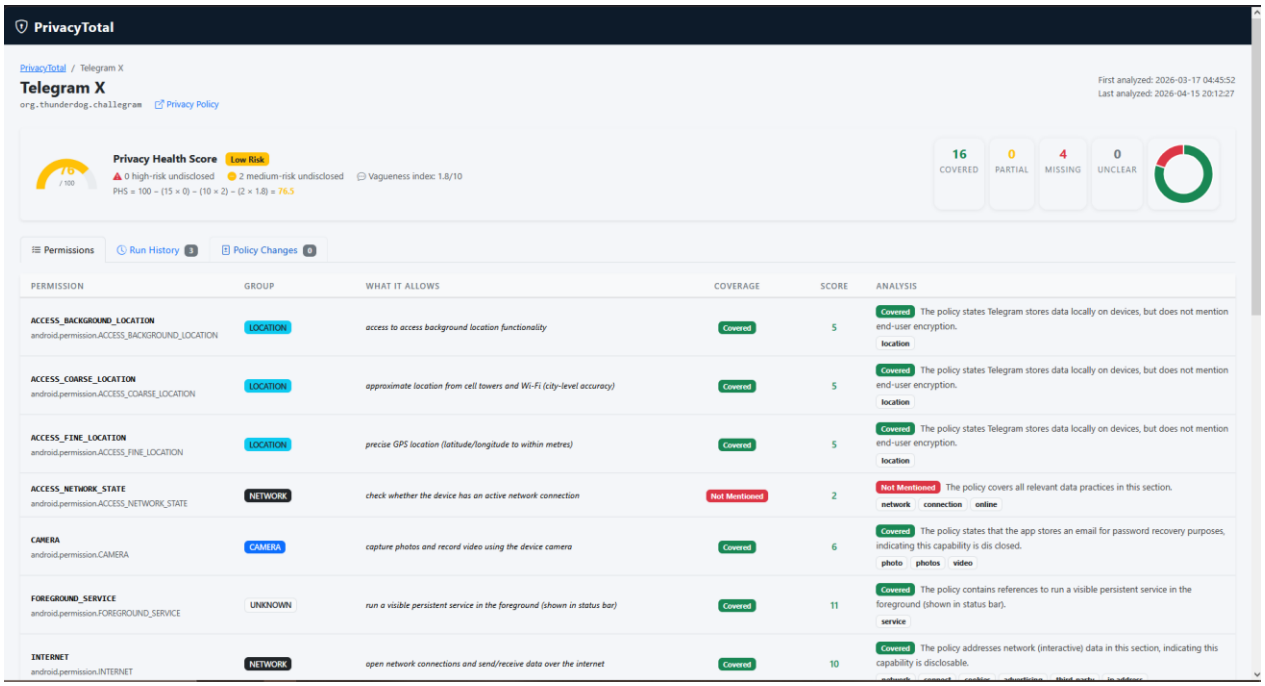


Figure 3.8 - Application detail page. The reader should note the three synchronised views of the same run: the coverage donut, the PHS gauge, and the per-permission table. Together they show which specific permissions drive the overall score.

Implementation

Development Environment

Operating System	Windows 10 Pro
Python Version	3.11
GPU	RTX 2060 Super NVIDIA GPU (6GB VRAM minimum for QLoRA training)
Key Libraries	PyTorch, Transformers, PEFT, BitsAndBytes, Playwright, BeautifulSoup4, Flask
Database	SQLite 3 (via ‘sqlite3’ standard library)

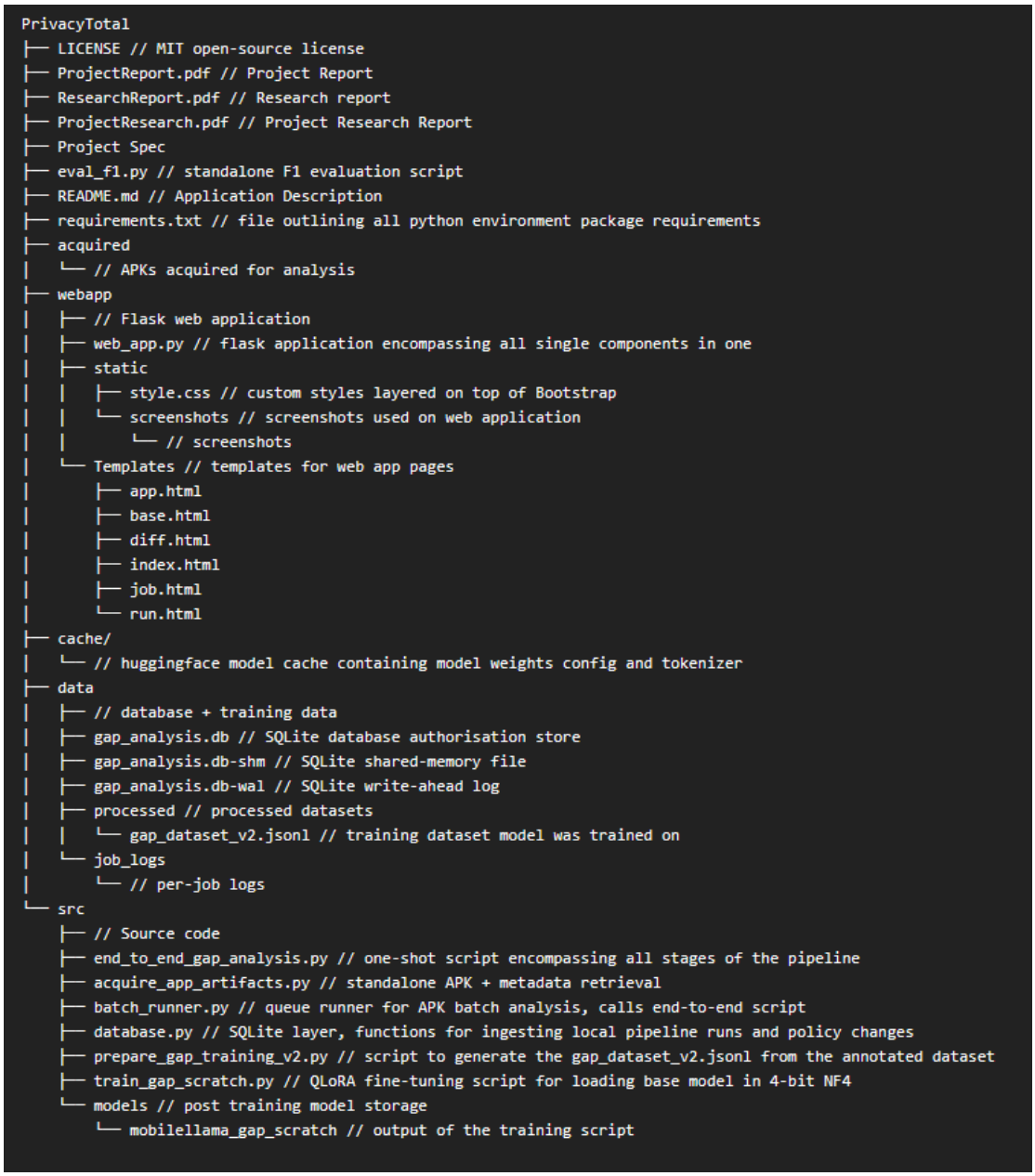


Figure 4.1 - Project directory structure. The reader should note the separation between 'src/' (pipeline and training scripts), 'data/' (raw and processed training corpora), 'webapp/' (Flask application), and 'src/acquired/' (per-run output directories), which is the boundary used throughout the report when referring to artefacts.

APK Acquisition: Five-Source Fallback Chain

The APK acquisition module ('src/end_to_end_gap_analysis.py') implements the fallback chain through five separate fallback websites. Key details for implementation include:

APKMirror uses a two-stage approach. First, a direct URL is constructed from a precomputed developer-slug mapping (approximately 60 entries in '_DEV_SLUG_MAP'), which bypasses the Cloudflare-protected search page for known developers. For unknown applications, Playwright runs an automated headless browser session and captures the CDN download URL from network response events rather than scraping anchor tags, which ensures that the correct direct-download URL is obtained even when the URLs served by the CDN are dynamically generated.

The user-agent pool cycles through nine distinct browser fingerprints across retry attempts (maximum seven retries with exponential back-off) to reduce the risk of bot detection.

Uptodown uses a chain of token-extraction patterns to retrieve the 'data-url' attribute required to construct the CDN download URL:

- Extract from '[data-button-id="detail-download-button"]'
- Extract any 'data-url' attribute ≥ 20 characters
- Extract a 'dw.uptodown.com' href directly

Additionally, if the '/download' page presents a version list rather than a direct download, the code follows a version-specific URL pattern ('/android/{version}/download').

Post-download package name verification reads the 'AndroidManifest.xml' file directly from the APK ZIP archive and checks for the expected package name as both ASCII bytes and UTF-16-LE encoded bytes (covering both the standard binary XML encoding and Unicode encoding variants).

Training Data Preparation

The training dataset ('data/processed/gap_dataset_v2.jsonl') was generated by 'src/prepare_gap_training_v2.py'. The script produces 5,154 examples from three sources:

Source 1 - Synthetic permission template: For each of the 20 most Android permissions, a set of positive (covered) and negative (not mentioned) example pairs was hand-authored. Each example uses a realistic policy excerpt and a corresponding single-line analytical response beginning with 'Mentioned:' or 'Not Mentioned:'. This provides a balanced, high-quality supervision signal for all major permission groups.

Source 2 - Real acquired data: The privacy policy and permissions from a manually acquired application ('com.miniclip.plagueinc') were used to generate real examples, with automatic excerpt retrieval and deterministic labelling providing accurate labels to compare.

Source 3 - PrivacyQA remapping: The PrivacyQA training corpus was remapped to the permission-style capability format. Questions from PrivacyQA were mapped to permission

group queries, and question/answer pairs were reformatted into the ‘Capability: Group (SHORT_NAME) / Keywords:’ prompt format with analytical one-line responses replacing the original policy quotes.

All examples use the exact prompt format that the inference pipeline constructs at runtime:

Instruction:

{instruction_header}

Capability: {group} ({short_name})

Keywords: {keywords}

EXCERPT:

{policy_excerpt}

Response: {analytical_response}

Alignment between the training and inference prompt formats is critical: an earlier training run that used a slightly different prompt format produced severely degraded outputs at inference, as discussed in Section 6.

Dataset Composition and Representativeness

Of the 5,154 training examples, approximately 60% (Source 1) are synthetic hand-authored permission templates, approximately 5% (Source 2) are drawn from a single manually acquired application (‘com.miniclip.plagueinc’), and approximately 35% (Source 3) are reformatted PrivacyQA question-answer pairs. The dataset is therefore heavily weighted toward synthetic supervision, and the reader should treat it as such.

This composition introduces three known biases. First, synthetic examples tend to use clearer and more formulaic language than real-world privacy policies, so the model may learn to reward textbook phrasing and penalise genuinely vague but valid disclosures. Second, coverage is concentrated on the 20 most common permission groups (LOCATION, CAMERA, MICROPHONE, CONTACTS, STORAGE, PHONE, NOTIFICATIONS, NETWORK, BILLING, and related), which leaves long-tail permissions such as ‘BODY_SENSORS’, ‘PROCESS_OUTGOING_CALLS’, and ‘NFC’ under-represented. Third, the analytical-response format was designed by the author, so the model may reproduce stylistic patterns of that author rather than of a legal reviewer.

The evaluation set of 100 acquired applications does not overlap with the single application used in Source 2, and no Google Play package ID appears both in training and in evaluation. PrivacyQA policies (Source 3) were used only to supply textual excerpts for instruction-following

examples; they were not drawn from the same apps used in evaluation. No hold-out validation split was used during training because the evaluation is an end-to-end pipeline assessment on fresh applications rather than a held-out portion of the training corpus, and the training loss alone is not claimed as evidence of generalisation.

Model Fine-Tuning

The model was fine-tuned using the configuration in 'src/train_gap_scratch.py'. The base model was 'mtgv/MobileLLaMA-2.7B-Chat', loaded in 4-bit NF4 quantisation via BitsAndBytes.

LoRA Configuration:

- Rank r = 8 (conservative, to preserve base instruction-following capability)
- Alpha = 16 // Scaling factor for LoRA updates; balances how strongly the model is influenced by the weights
- Dropout = 0.05 // Prevent's overfitting during fine-tuning process
- Target modules: 'q_proj', 'v_proj', 'k_proj', 'o_proj' // Applies LoRA to the mentioned attention layers (query, value, key, output projections) which is important for learning

Training Configuration

- Epochs: 3 // Number of full passes over the dataset
- Per-device batch size: 2; gradient accumulation steps: 8 (effective batch = 16) // smaller batch step due to VRAM limits, where accumulation simulates a larger batch
- Learning rate: 5×10^{-5} // conservative rate as earlier runs at 2×10^{-4} caused catastrophic forgetting
- LR scheduler: cosine decay with 10% warmup // Gradually increases LR at start to 'warmup' then smoothly decays over time to stabilise training
- Max gradient norm: 0.3 // Prevents unstable or overly large updates during training
- Optimiser: 'paged_adamw_8bit' // Efficient training method that reduces memory use
- Sequence length: 512 tokens // Limits the amount of text processed in a single training step
- Seed: 42 // Ensures results of training can be reproduced consistently

Training ran for approximately 2.5 hours on the RTX 2060 Super with loss dropping from 2.91 to 0.34. The final checkpoint at 'src/models/mobilellama_gap_scratch/' is the production model used.

Two earlier training phases - Phase 1 for Princeton-Leuven domain adaptation and Phase 2 for OPP-115 fine-tuning - were stacked as LoRA adapters beneath the Phase 3 classification training. Phases 1 and 2 had taught the model to generate summary-style privacy-policy text rather than to classify whether a disclosure was present. When Phase 3 classification training was applied on top of these adapters, the model exhibited catastrophic forgetting: it ignored instructions in

the prompt and produced hallucinated text resembling policy boilerplate. The resolution was to discard the Phase 1 and Phase 2 adapters and retrain from the base model on a single smaller classification-style dataset.

Inference Pipeline

At inference time, the ‘load_model()’ function loads the base model in 4-bit quantisation and merges the ‘mobilellama_gap_scratch’ LoRA adapter. Key configuration decisions were:

- ‘prompt_max_length = 2048’: The earlier value of 1024 produced left-truncation that removed the ‘Instruction:’ header, causing the model to emit unstructured output. Increasing the budget to 2048 preserves the header.
- ‘truncation_side = "right"’: Left truncation was incorrectly removing the instruction header. Right truncation preserves the instruction and truncates only the policy excerpt when necessary.
- ‘no_repeat_ngram_size = 3’: A value of 6 was too aggressive for a 2.7B model on short responses and caused premature end-of-sequence (EOS) tokens. Reducing to 3 allowed the model to complete the response.
- ‘do_sample = False’ (greedy decoding): Greedy decoding ensures deterministic, reproducible outputs, which is necessary for auditing and for regression-testing the pipeline.

The ‘clean_model_response()’ function extracts the first generated line, strips the prompt prefix if it is echoed, normalises variant spellings of ‘Mentioned:’, and validates a minimum response length of 30 characters. Raw model output is logged to ‘debug_last_model_output.txt’ before any cleaning, enabling post-hoc debugging.

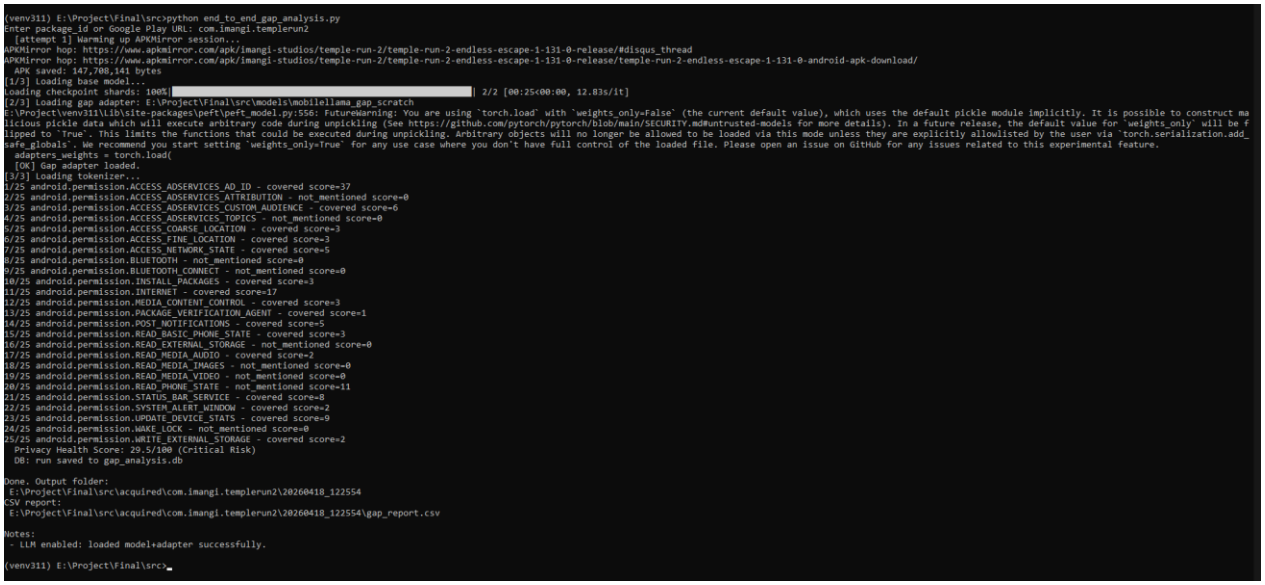


Figure 4.3 - End-to-end CLI run for a single application. The reader should note the order of operations: APK acquisition (with source fallback), permission extraction, policy fetch, per-

permission LLM classification, and final PHS computation - all timed, and all writing artefacts under the same timestamped output directory.

Privacy Health Score Algorithm

The Privacy Health Score (PHS) is a heuristic scalar metric on a 0-100 scale that summarises an application's overall disclosure completeness. It is computed as:

PHS = 100 x (covered_count + 0.5 x partially_covered_count) / total_count

A Vagueness Index is also computed as the fraction of permissions classified as 'partially_covered':

Vagueness = partially_covered_count / total_count

Applications with zero analysable permissions (e.g., no APK obtained) receive a 'NULL' PHS rather than an inaccurate score. Risk labels are assigned to PHS ranges:

- 80 - 100: Low Risk
- 60 - 79: Moderate Risk
- 40 - 59: High Risk
- 0 - 39: Critical Risk

Web Application

The PrivacyTotal web application ('webapp/web_app.py') is a Flask application with five Jinja2 templates and five route handlers.

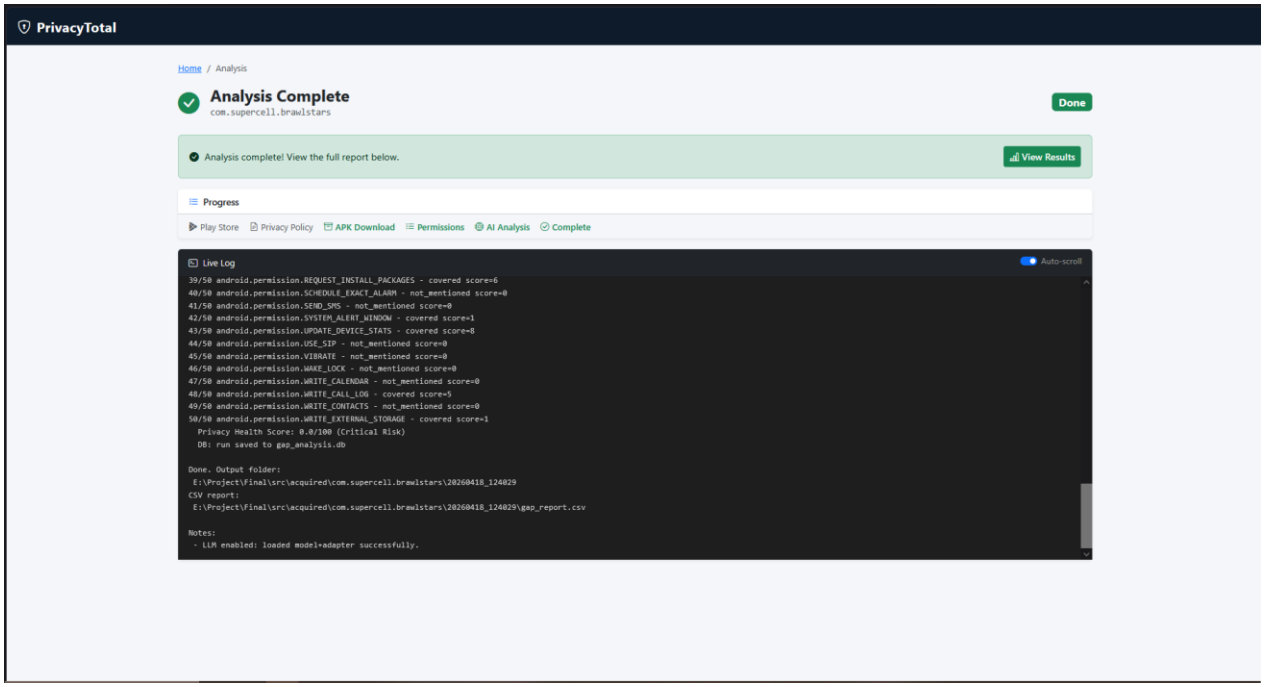
Routes:

- 'GET /' - main page with three tabs (Analyse App, Browse Database, Research Report)
- 'POST /analyse' - accepts a package ID form submission, creates a background job and redirects to the job status page
- 'GET /job/<job_id>' - live job status page, polls 'GET /job/<job_id>/status' for completion
- 'GET /app/<package_id>' - application details page
- 'GET /diff<package_id>' - policy change page showing different policies per application

The database module's 'ingest_acquired_dirs()' function is called when the application is started to ingest any analysis runs performed via the command-line pipeline ('end_to_end_gap_analysis.py') into the database to ensure consistency between the databases.

The ‘sort_phs’ custom Jinja2 filter is used to ensure that applications with a PHS value of ‘None’ sort to the end of the Browse Database table, preventing these values from sorting to the top or bottom unexpectedly.

URL hash deep-linking ‘/?#tab-browse', ‘/?#tab-report') is created via JavaScript's ‘hashchange’ listeners which allows the application to be bookmarked to a tab for user convenience.



Figur e 4.4 - Web-app job status page during a live run, current example showing an application finishing it’s analysis.

Key Technical Challenges and Solutions

Challenge	Root Cause	Solution
Playwright ‘event loop already running’ error on Windows	Python 3.11 asyncio + Playwright sync API incompatability	‘nest_asyncio.apply()’ at import time
Model generates hallucinated text instead of classifications	Forgetting of Phase 1 & 2 adapters when training Phase 3 with high Learning Rate	Retrained from base model with single script instead of training pipeline, reduced learning rate
LLM instruction header cut off due to truncation	‘prompt_max_length=1024’ truncating the models instruction prompt	Increased to 2048 and switched to right truncation to truncate from the right
Computer switching off during model training	GPU fan malfunctioning causing the GPU to overheat and shut off	Temporarily replaced with 6gb VRAM replacement GPU while repairing GPU

Uptodown CDN returning wrong APK's	CDN routing error for some high profile apps (Roblox returned Instagram)	Post-download package name verification from the AndroidManifest.xml file
APKMirror Cloudflare bot detection	Standard headless chromium fingerprint was detected and blocked	10-signal TLS fingerprint used to mask Javascript

Testing and Evaluation

Testing Methodology

The evaluation strategy combines three levels of testing:

- Unit Testing: Individual pipeline components were tested prior to being put together in the 'end_to_end_gap_analysis.py' script
- Integration Testing: Assembling the entire end to end pipeline and testing on known applications
- Model Evaluating: Comparing the LLM classification output against the verified ground truth labels across 100 applications.

Manually reviewing each permission-policy pair across the 100 evaluation applications was not feasible within the project timeline. A deterministic keyword oracle was therefore used as a proxy ground truth: pairs with a keyword score of at least 3 were labelled 'covered', pairs with a score of 0 were labelled 'not_mentioned', and ambiguous pairs (scores 1-2) were excluded from the F1 computation. This provided a consistent deterministic signal against which the LLM's predictions were compared to compute Precision, Recall, F1, and macro-averaged metrics. The reported scores should be read as agreement with this proxy, not as agreement with expert annotation; Section 5.4 discusses the resulting threats to validity.

Unit and Integration Testing

Permission Extraction: Verified against eight applications in the 'src/acquired/' directory, comparing extracted permissions against permissions visible in the Google Play listing and known APK analysis tools. All declared '<uses-permission>' entries were correctly extracted.

Policy Retrieval: Tested across 100 applications where policy retrieval was successful for all applications with publicly accessible policy URLs. The HTML-to-text extraction was validated by manually reviewing 'policy.txt' against the source policy web page for ten applications.

PHS Computation: Edge cases verified the following:

- Empty permission list gives the PHS a 'None' value

- All covered means the PHS = 100
- All not_mentioned means the PHS = 0
- Mixed results will provide a correctly weighted average

Web Application: Manual testing of all routes, form submissions, job creations and status pollings. Policy diff page was tested against applications with policy text changes, such as 'Soundcloud', which patched a non-disclosure of a high-risk permission in its policy.

A lightweight test script ('quickcheck.py') is present in the acquired directory for Clash of Clans ('com.supercell.clashofclans') and validated that the gap reports are outputting columns and classifications in the correct format.

Model Evaluation

The production model ('mobilellama_gap_scratch') was evaluated on all permissions across 117 independently acquired Android applications. Runs with zero permissions (no APK was obtained) were excluded automatically by 'eval_f1.py'.

Overall Results (117 applications, 175 analysis runs, 3483 permission-policy pairs, 2562 oracle-usable pairs):

Metric	Score
Precision	0.892
Recall	0.971
F1 Score	0.930
Macro-Average F1	0.906
Weighted-Average F1	0.911
LLM Agreement Rate	92.5 %

Coverage Distribution:

- Covered: 65.2% (2272 permissions)
- Not Mentioned: 25.7% (895 permissions)
- Unclear / Partially Covered: 9.1% (316 permissions)

Comparison to Prior Training Approaches:

Earlier model versions (Phase 3 on stacked Phase 1 and Phase 2 adapters) achieved near-zero useful classification - most outputs were hallucinated policy-style text rather than one of the expected labels. The retraining-from-scratch approach, combined with a format-consistent dataset and conservative hyperparameters, produced the F1 of 0.930 in a fraction of the

training time consumed by the failed runs, suggesting that dataset quality and prompt-format consistency were more important than dataset size for this task.

Threats to Validity

The headline Precision of 0.892 and F1 of 0.930 on the covered class must be interpreted with several caveats:

Proxy-label evaluation, not expert annotation was used: The ground-truth labels against which the LLM was scored were generated by a deterministic keyword oracle, not by human privacy experts or legal reviewers. The reported scores therefore measure agreement between the LLM and the keyword oracle, not agreement between the LLM and a domain expert. A higher F1 against the oracle does not by itself imply that the model makes sound disclosure judgements.

Circularity between the oracle and the pipeline: The keyword evidence used by the oracle comes from the same 'PERM_KEYWORDS' mapping that the pipeline uses to (i) retrieve policy excerpts, (ii) pre-classify permissions with no keyword hits as 'not_mentioned', and (iii) determine the final coverage label in combination with the LLM response. Because the LLM is shown excerpts selected by the same keyword signal that drives the oracle, there is a structural risk that the evaluation rewards the model for agreeing with an input it has been biased toward. The reported scores should therefore be read as an upper bound on true performance.

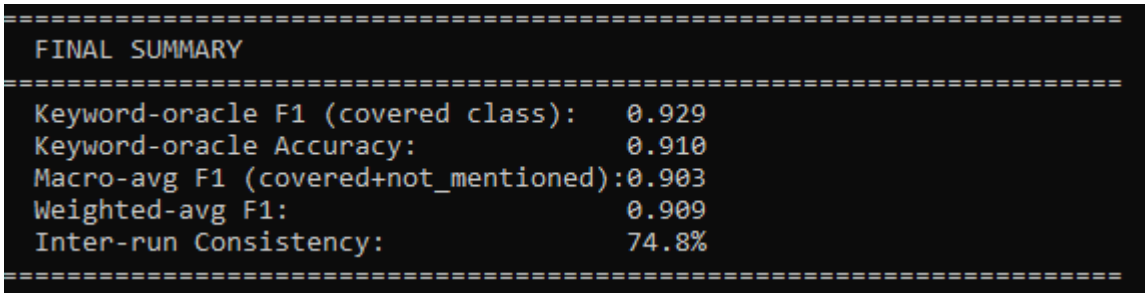
Exclusion of ambiguous cases: Pairs with intermediate keyword scores (1-2) were excluded from F1 computation. These are the pairs most likely to involve genuinely vague policy language - exactly the cases where expert human judgement adds the most value. Excluding them removes the hardest slice of the data from the evaluation and biases the reported scores upward. The 9.1% of pairs labelled 'unclear' in the coverage distribution represent this difficult middle band.

Vague-disclosure false positives: A policy sentence that contains a keyword but does not describe a concrete data practice (for example, 'we may use information') satisfies both the oracle and the prompt, so both the oracle and the LLM can agree that a permission is 'covered' even when a human reader would judge the disclosure inadequate.

Synthetic training data shaping the decision boundary: Because approximately 60% of the training set is synthetic (see Section 4.3), the model may have learned stylistic cues present in the synthetic examples rather than general semantic criteria for adequate disclosure. Classification accuracy on genuinely distributional policy text may therefore be lower than the proxy-based F1 suggests.

A stronger evaluation, out of scope for this project, would commission a small set of expert-annotated permission-policy pairs (on the order of 200-500) and report both proxy F1 and

expert-agreement F1 side by side. In the absence of such a gold standard, the results in this section should be read as: the model is highly consistent with the deterministic signal used by the rest of the pipeline.



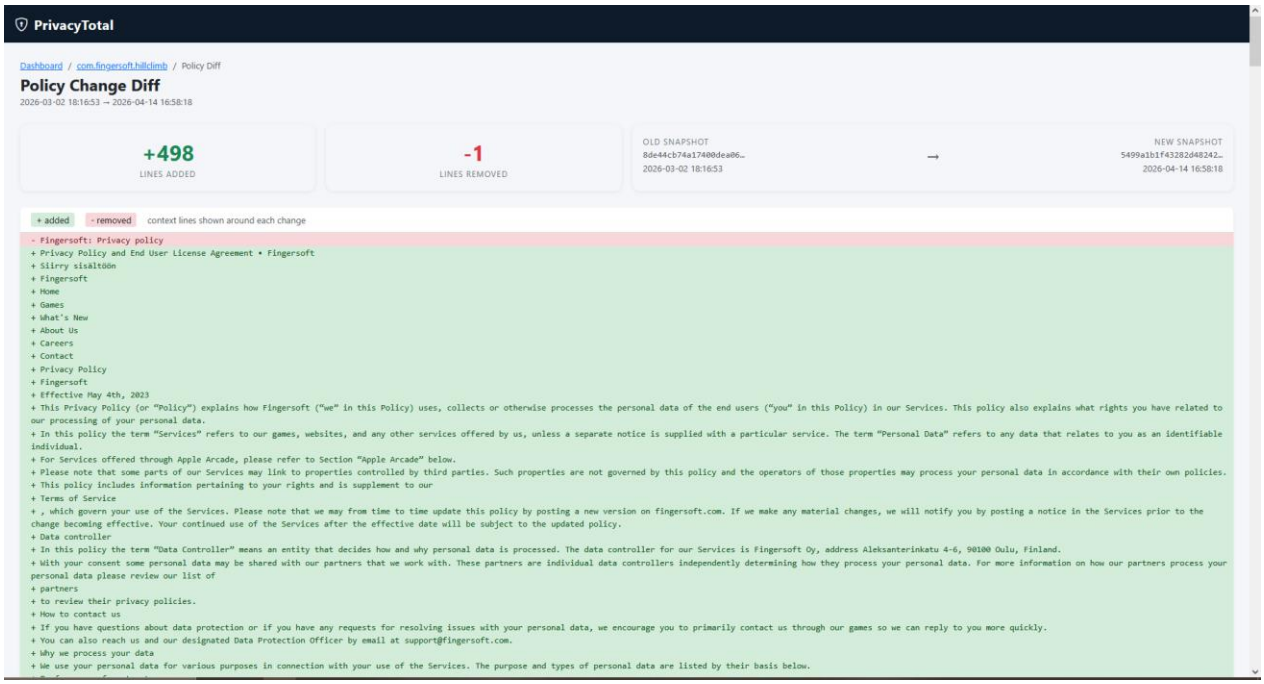
Figure(s) 5.1 - Model evaluation output from 'eval_f1.py'. The reader should note the scores alongside Section 5.4 on threats to validity: these values measure agreement with the deterministic keyword oracle rather than with expert annotation.

System Level Evaluation

APK Acquisition Success Rate: APKs were successfully downloaded for all 117 applications retained in the analytical corpus. Candidate applications for which all five download sources failed were dropped during pre-screening and replaced with alternates from the same category; these were typically very new releases or region-restricted applications.

Pipeline Throughput: End-to-end analysis of a single application (APK download, permission extraction, policy retrieval, LLM inference for all permissions) takes around 3-8 minutes, but this is largely dependent on the size of the application, number of permissions and network conditions.

Database Consistency: 'ingest_acquired_dirs()' correctly imports all runs into the database at application startup, and 'policy_change_pairs()' correctly identifies the seven applications whose policy text changed between runs.



Figure(s) 5.2 - Policy diff view for an application whose policy text changed between runs. The reader should note the inline colour-coded additions and removals, which allow a reviewer to identify the exact sentences that were added, modified, or removed, rather than having to re-read the whole policy.

Case Studies

Temple Run 2 ('com.imangi.templerun2'): This casual endless runner game declares 25 permissions and the gap analysis identified 16 as 'covered' and 9 as 'not_mentioned', mostly revolving around adservices, bluetooth and storage permissions. This wide gap in disclosures awards the application with a PHS of 30 (Critical Risk).

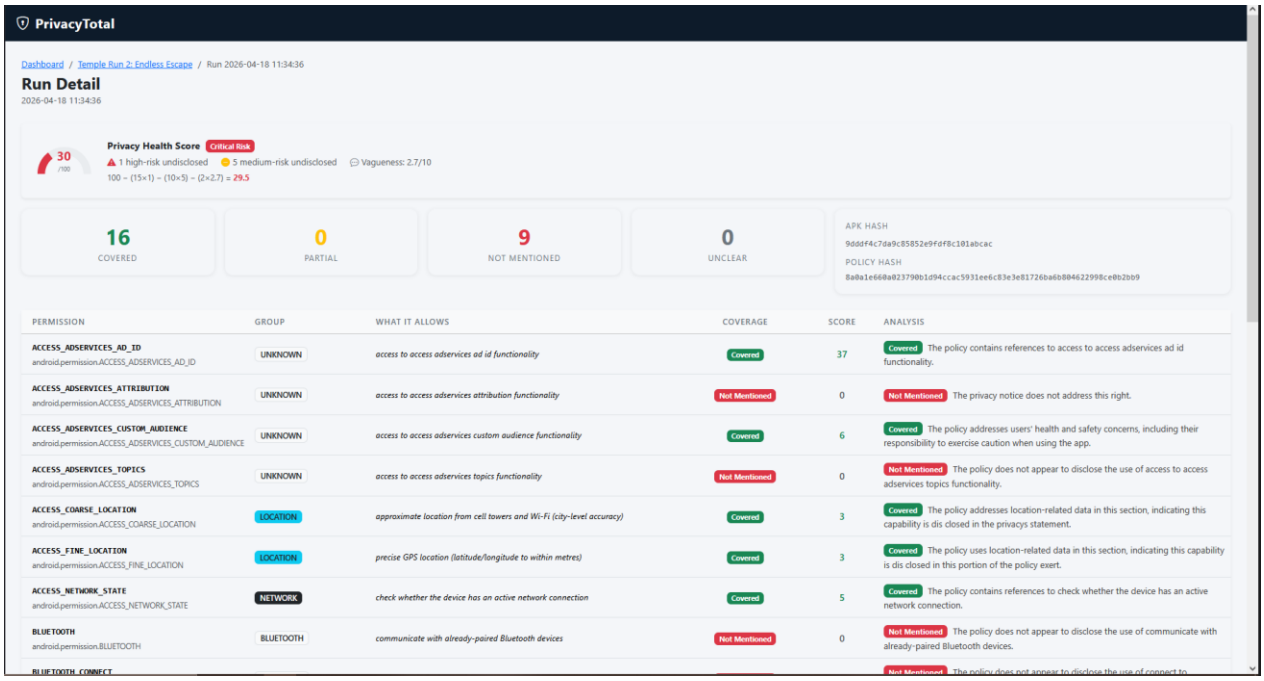


Figure 6.1 - Temple Run 2 application detail page. The reader should note the concentration of 'not_mentioned' labels around advertising, Bluetooth, and storage permissions, which is the pattern driving the 29.5 PHS score (Critical Risk).

Among Us ('com.innersloth.spacemafia'): This multiplayer game declares 12 permissions. Gap analysis identified 5 discrepancies, primarily Bluetooth, NFC, and Storage permissions that lacked clear disclosure in the policy text despite being declared in the APK 'AndroidManifest.xml' file. The resulting PHS was 53 (High Risk).

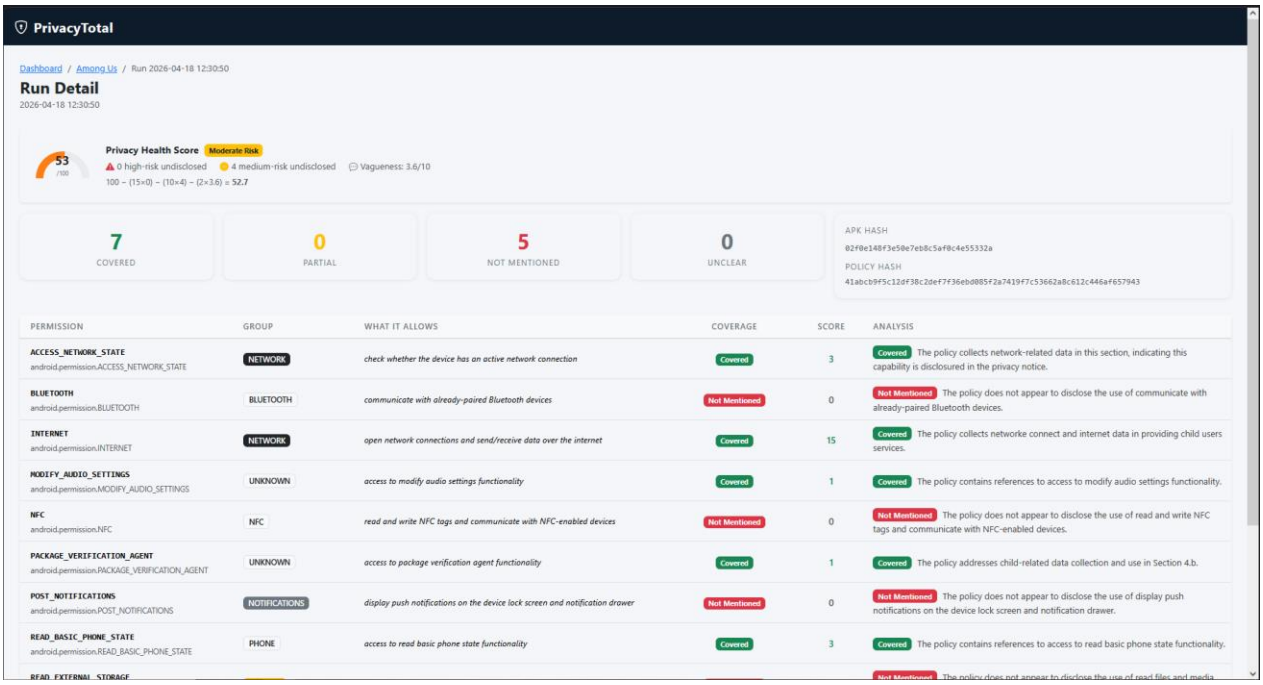


Figure 6.2 - Among Us application detail page.

SoundCloud ('com.soundcloud.android'): This music application declared 15 permissions, of which only 2 (location and vibration) were undisclosed in the policy, giving it a PHS of 93 (Low Risk).

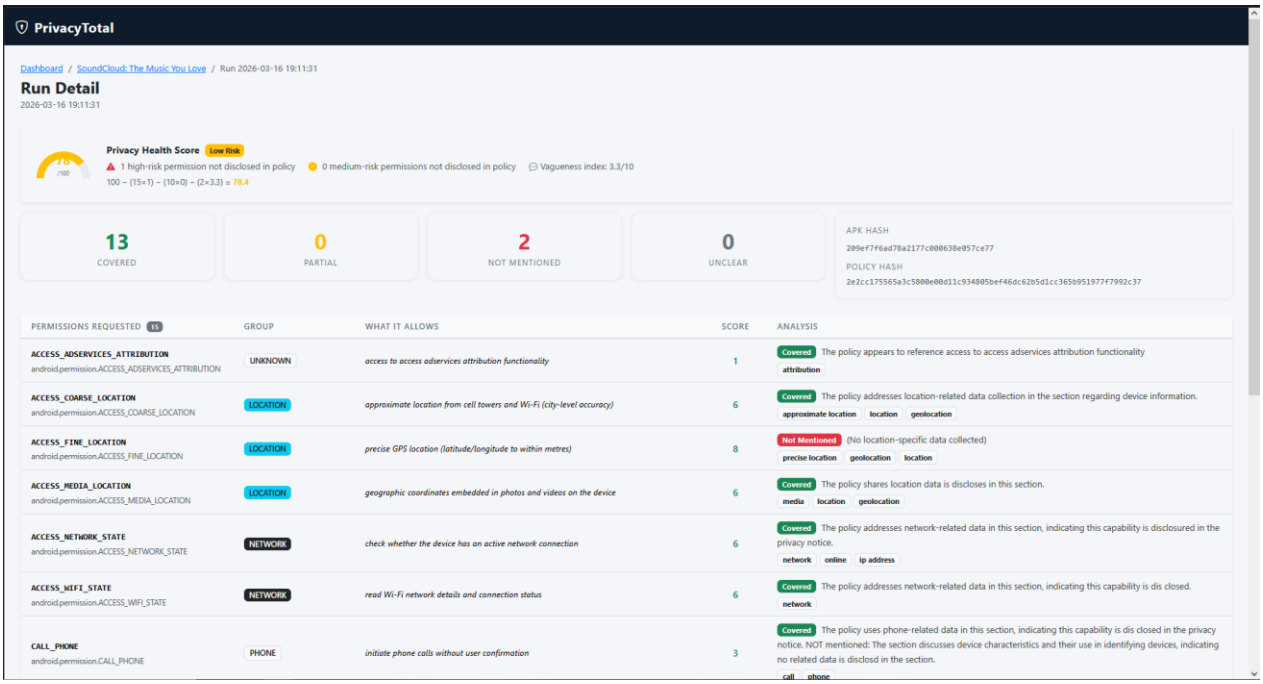


Figure 6.3 - SoundCloud application detail page. The reader should note that only two permissions are flagged as undisclosed, which is the pattern that places the application in the

Low Risk band; this is also the app used to validate the policy-diff feature, because its policy text changed between runs.

Discussion

Challenges Faced

Bot Detection and Rate Limiting: The primary ongoing challenge was obtaining APK files without triggering bot detection on mirror sites. APKMirror's Cloudflare integration was particularly difficult to work around, and the final solution required JavaScript fingerprint masking across ten different browser signals. Some applications with high download volumes (WhatsApp, Instagram, TikTok) required per-application manual investigation of the download flow before the process could be automated.

Catastrophic Forgetting in Sequential Training: The most significant technical failure of the project was the Phase 3 model collapse caused by catastrophic forgetting. Several weeks of training time were lost while diagnosing the root cause, which was a combination of poorly formatted training data (verbatim policy text used as responses rather than analytical classifications), an overly high learning rate, and training defaults that were carried over between attempts. Recovery required restarting from the base model with a smaller, format-consistent dataset, which simplified the architecture and produced substantially better results.

Prompt Engineering Weaknesses: The inference pipeline is sensitive to the exact prompt format used during training. Early inference runs used a slightly different capability description format (e.g., 'Capability: {permission_name}' rather than 'Capability: GROUP (SHORT_NAME)') and produced significantly degraded output as a result. Aligning the inference prompt format exactly to the training data format was critical in fixing this gap.

Multi-source fallback chains require end-to-end verification: The post-download APK package-name check prevented silent errors where a CDN served the wrong application's APK. Without this check, the pipeline would have processed permissions from one application while reporting results for another, and the error would have been detectable only through manual spot-checks.

Conservative hyperparameters are safer when base-model capability is valuable: MobileLLaMA 2.7B already possesses strong instruction-following behaviour, so high learning rates, large LoRA ranks, and long training runs all contributed to erosion of the base model's general competency. Lowering these values produced substantially better downstream results.

Comparison to Original Objectives

All eight original project objectives were achieved, and the minimum application-count and F1 targets were exceeded. The project scope expanded during implementation to include:

- Policy change detection across multiple runs
- Privacy Health Score as a consumer-facing summary metric
- Vagueness Index as a secondary metric
- Five-stage APK download fallback chain

The one original objective that was set aside was producing long, detailed policy summaries. This was replaced with a concise coverage label and a brief excerpt, on the basis that a short, structured output is easier for consumers to interpret quickly than a long summary.

Lessons Learned

Training data quality outweighs quantity: The 'gap_dataset_v2.jsonl' dataset (5,154 examples) produced substantially better results than the earlier 'phase3_gap_dataset.jsonl' dataset of 16,500 examples, because the v2 examples use short, analytical, format-consistent responses. More training data with inconsistent response format is harmful; less data with consistent format is beneficial.

Format alignment before content alignment: The model's ability to produce correctly formatted outputs ("Mentioned: ..." / "Not mentioned: ...") depended entirely on training responses being in exactly that format. Accurate content followed once the format was correct.

Deterministic fallback before LLM is cost-effective: Keyword-based pre-classification before LLM inference removes approximately 25% of calls by immediately labelling permissions with zero keyword hits as 'not_mentioned'. For a 2.7B model running on a low-VRAM GPU, the resulting latency reduction is meaningful.

Future Work

Expanded Model Coverage

The current model was trained on 5,154 examples spanning 20 permission groups. A larger and more refined training dataset, incorporating the entire PrivacyQA corpus, additional acquired applications, and examples generated by stronger teacher models, could improve recall on edge cases, particularly for niche permissions such as 'BODY_SENSORS', 'PROCESS_OUTGOING_CALLS', and 'NFC'.

Multi-Language Policy Support

All policies in the current evaluation are English-language. Extending the pipeline to support multilingual policies - particularly German, French and Spanish given GDPR jurisdictions - would significantly broaden applicability. This could be achieved by replacing MobileLLaMA with a multilingual base model (e.g., Mistral 7b with multilingual fine-tuning) or by adding a translation pre-processing step.

GDPR Compliance Mapping

Beyond binary disclosure checking, future work could map each permission's data practice to specific GDPR requirements (Article 13 disclosure obligations, lawful-basis requirements, data-retention statements). This would extend PrivacyTotal from a gap-detection tool into a structured GDPR compliance-assessment tool.

Batch Analysis at Scale

The current batch runner ('batch_runner.py') operates sequentially, analysing one application at a time. For large-scale research (for example, analysing the top 10,000 Play Store applications), a distributed task queue would enable parallel processing across multiple machines.

Mobile Application

A mobile companion application could allow users to point their phone at an application's Play Store listing and receive an instant Privacy Health Score, making the tool more accessible.

Conclusion

This project designed, implemented, and evaluated PrivacyTotal: an end-to-end automated system for identifying privacy disclosure gaps in Android applications. Starting from a Google Play package identifier alone, the system acquires the APK, extracts declared permissions, retrieves the privacy policy, and uses a fine-tuned language model to classify each permission as covered, not mentioned, or partially covered in the policy.

The core technical contributions are:

- A five-source APK acquisition pipeline with anti-bot countermeasures and post-download package-name verification, evaluated across 117 applications (Section 5.5).
- A domain-specific LoRA fine-tuned model (MobileLLaMA-2.7B-Chat) trained from scratch on 5,154 permission-disclosure examples, which achieved an F1 of 0.930 against a deterministic keyword proxy (Section 5.3), exceeding the project minimum target of 0.85.
- A complete web application exposing the pipeline through a consumer-facing interface with Privacy Health Scores, run history, and policy-change detection (Section 4.6).

The evaluation shows strong agreement between the model and the deterministic oracle used by the rest of the pipeline. As discussed in Section 5.4, this result should be read as evidence that the system behaves consistently against a proxy benchmark; it does not establish agreement with expert legal or privacy review. The project therefore positions PrivacyTotal as a scalable triage tool that surfaces candidate disclosure gaps for downstream human review, rather than as a substitute for legal compliance assessment.

The training failures , in particular the catastrophic forgetting observed when classification training was stacked on top of summarisation-style adapters, demonstrate that LoRA fine-tuning is more sensitive to training-data format and learning rate than was initially assumed, and that dataset quality and format-consistency outweigh dataset size for this task.

In summary, PrivacyTotal is a strong research prototype for automated privacy-disclosure gap detection. Stronger validation, particularly a panel of expert-annotated permission-policy pairs and a broader evaluation across jurisdictions and languages, would be required before any claim of compliance-grade accuracy could be justified. Within its evaluated scope, the system demonstrates that a small quantised LLM on commodity hardware can produce consistent, reproducible, per-permission disclosure classifications together with a summary score suitable for consumer-facing use.

References

Barrera, D., Kayacik, H.G., van Oorschot, P.C. and Somayaji, A. (2010) 'A methodology for empirical analysis of permission-based security models and its application to Android', in Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS), pp. 73-84.

McDonald, A.M. and Cranor, L.F. (2008) 'The cost of reading privacy policies', Journal of Law and Policy for the Information Society, 4(3), pp. 540-565.

Wilson, S., Schaub, F., Dara, A.A., Liu, F., Cherivirala, S., Leon, P.G., Andersen, M.S., Zimmeck, S., Sathyendra, K.M., Russell, N.A., Norton, T.B., Hovy, E., Reidenberg, J. and Sadeh, N. (2016) 'The creation and analysis of a website privacy policy corpus', in Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL), pp. 1330-1340.

Zimmeck, S., Wang, Z., Liu, L., Adjerid, I., Story, P., Smullen, D., Schaub, F., Sadeh, N., Bellovin, S.M. and Reidenberg, J. (2017) 'Automated analysis of privacy requirements for mobile apps', in Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS).

Story, P., Zimmeck, S., Ravichander, A., Smullen, D., Wang, Z., Reidenberg, J., Russell, N.A. and Sadeh, N. (2019) 'Natural language processing for mobile app privacy compliance', in Proceedings of the AAAI Spring Symposium on Privacy-Enhancing AI and Language Technologies.

Andow, B., Mahmud, S.Y., Wang, W., Whitaker, J., Enck, W., Reaves, B., Singh, K. and Xie, T. (2019) 'PolicyLint: investigating internal privacy policy contradictions on Google Play', in Proceedings of the 28th USENIX Security Symposium, pp. 585-602.

Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K. (2019) 'BERT: pre-training of deep bidirectional transformers for language understanding', in Proceedings of NAACL-HLT 2019, pp. 4171-4186.

Felt, A.P., Chin, E., Hanna, S., Song, D. and Wagner, D. (2011) 'Android permissions demystified', in Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS), pp. 627-638.

Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C. and Amodei, D. (2020) 'Language models are few-shot learners', Advances in Neural Information Processing Systems, 33, pp. 1877-1901.

OpenAI (2023) GPT-4 Technical Report. arXiv preprint arXiv:2303.08774.

Srinath, M., Wilson, S. and Giles, C.L. (2021) 'PrivBERT: privacy policy classification with domain-specific pretraining', in Proceedings of the 2021 Workshop on Natural Language Processing for Privacy (PrivNLP).

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E. and Lample, G. (2023) LLaMA: Open and Efficient Foundation Language Models. arXiv preprint arXiv:2302.13971.

Zhang, X. et al. (2024) 'MobileLLaMA: an instruction-following mobile-optimised language model', in Proceedings of ACL 2024.

Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L. and Chen, W. (2021) 'LoRA: low-rank adaptation of large language models', in Proceedings of ICLR 2022.

Dettmers, T., Pagnoni, A., Holtzman, A. and Zettlemoyer, L. (2023) 'QLoRA: efficient finetuning of quantized LLMs', Advances in Neural Information Processing Systems, 36.