TLS Posture Analyzer: Automated Android TLS Security Auditing Powered by Artificial Intelligence

Baba Wiam[1], Bagy Oussama[1], Elaantri Malika[1], Ouzanzoul Bouchra[1]

ENSA Marrakech, Université Cadi Ayyad, Morocco

Abstract

Android applications routinely expose sensitive data by misconfiguring TLS, accepting self-signed certificates, or relying on broken trust-chain mechanisms. Manual auditing of Android Package (APK) files is slow, error-prone, and difficult to reproduce at scale. This paper presents **TLS Posture Analyzer**, an open-source automated pipeline for Android TLS security auditing. The platform integrates a pure-Python Android Binary XML (AXML) decoder, a three-pass `jadx` decompilation strategy resilient to bytecode obfuscation, and 17 Java security-pattern detectors covering CWE-295, CWE-297, CWE-326, CWE-327, and OWASP Mobile Top 10 M2/M3. An AI reporting layer powered by the Groq API (model: `llama-3.3-70b-versatile`, `temperature=0.2`, `max_tokens=1024`) streams natural-language hardening recommendations in real time via Server-Sent Events (SSE). Code snippets from decompiled APKs are transmitted to the Groq cloud; analysts must hold a valid API key and should note that recommendations may vary between runs due to the non-deterministic nature of LLM inference. Built with FastAPI 0.115 for the back-end and React 18 for the front-end, the tool produces complete TLS posture reports with file-and-line source evidence. Its capabilities are demonstrated through two case studies: a walkthrough on the publicly available F-Droid APK covering all seven analysis stages, and a comparative evaluation against MobSF v4.5 using InsecureBankv2, where **TLS Posture Analyzer** identified transport-layer risks — including absent certificate pinning and implicit TLS configuration — not reported by the reference tool. The software is available at `https://github.com/M4l1k40/TLS-Posture-Analyzer` under the MIT licence. Current limitations include static-only analysis scope, sensitivity to heavy bytecode obfuscation, and cloud dependency for AI features.

Keywords: Android Security, TLS Misconfiguration, APK Static Analysis, Certificate Pinning, CWE, OWASP, Large Language Model, FastAPI, React, Automated Auditing

[1]École Nationale des Sciences Appliquées de Marrakech, Université Cadi Ayyad, Filière GCDSTE, Marrakech, Morocco. Supervised by Pr. Mohammed Lachgar.

Table 1: Required Metadata

Nr.	Code metadata description	Value
C1	Current code version	1.0.0
C2	Permanent link to code/repository	`https://github.com/M4l1k40/TLS-Posture-Analyzer`
C3	Permanent link to Reproducible Capsule	`https://github.com/M4l1k40/TLS-Posture-Analyzer/blob/main/README.md`
C4	Legal Code License	MIT License
C5	Code versioning system	Git
C6	Software languages, tools, and services	Python 3.11, FastAPI 0.115, Uvicorn 0.27, React 18.2, Vite 5.0, Node.js 18, jadx 1.5.0, apktool 2.9.3, Groq API (Llama 3.3 70B), HTML, CSS
C7	Compilation requirements & dependencies	Python 3.10+, Node.js 18+, npm, jadx, apktool (optional), Groq API key (optional); Windows/macOS/Linux. See `backend/requirements.txt` and `frontend/package-lock.json`.
C8	Link to developer documentation	`https://github.com/M4l1k40/TLS-Posture-Analyzer/blob/main/README.md`
C9	Support email	`lachgar.m@gmail.com`

1. Motivation and Significance

The security of mobile network communications is a pressing concern as Android applications increasingly handle sensitive personal, financial, and medical data [2]. The Android platform exposes a rich set of TLS configuration APIs — the Network Security Configuration (NSC) framework [1], custom `TrustManager` implementations, OkHttp `CertificatePinner`, and Retrofit builders — that developers frequently misconfigure. These defects are catalogued by MITRE as CWE-295 (Improper Certificate Validation), CWE-297 (Improper Validation of Certificate with Host Mismatch), CWE-326 (Inadequate Encryption Strength), and CWE-327 (Use of a Broken or Risky Cryptographic Algorithm) [7], and by OWASP as M2 and M3 in the Mobile Top 10 [4].

Large-scale empirical studies confirm the prevalence of these defects. Fahl et al. [2] found widespread certificate-validation failures in real-world Android applications, and Oltrogge et al. [3] demonstrated that the absence of certificate pinning exposes millions of users to man-in-the-middle attacks.

Gap in Existing Tools

Manual auditing requires a security analyst to decompile an APK, search through thousands of decompiled Java files, manually parse binary XML configurations, cross-reference proxy logs, and write a coherent risk report — a process that is slow, error-prone, and hard to reproduce. Existing tools address mobile security broadly but do not fill this specific gap. MobSF [8] provides comprehensive static and dynamic analysis but requires a complex deployment, relies on `apktool` as a hard dependency for NSC parsing, and produces no AI-generated natural-language report. QARK [9] targets general Android vulnerability detection with no dedicated TLS reporting layer. Drozer [10] focuses on dynamic exploitation and does not decompile APKs.

No existing open-source tool combines: (i) multi-pass decompilation resilient to obfuscation, (ii) purpose-built binary XML decoding without `apktool`, (iii) 17 pattern-specific TLS detectors with file-and-line evidence, and (iv) an AI-generated natural-language remediation report — all within a single, reproducible, low-dependency package.

Verifiable Software Contributions

1. A **pure-Python AXML decoder** (`backend/decompiler.py`) that reconstructs `network_security_config.xml` from compiled Android binary XML without invoking `apktool`.
2. A **three-pass `jadx` decompilation strategy** (`backend/decompiler.py`) with permissive-mode and partial-recovery fallbacks, improving robustness on obfuscated APKs.
3. A **library of 17 Java TLS security-pattern detectors** (`backend/analyzer.py`), each implementing a primary (presence) regex and a secondary (exploitability) regex, returning precise `{file, line, snippet}` evidence.
4. A **FastAPI + React 18 single-page application** that orchestrates the complete pipeline and streams AI-generated hardening recommendations via SSE.
5. A **batch REST endpoint** (`POST /analyze/folder`) enabling automated, large-scale APK corpus analysis.

What follows is organised around the tool's design and evaluation. Section 2 presents the architecture and end-to-end pipeline. Section 3 demonstrates the tool through a step-by-step analysis of a real APK. Section 4 examines where the tool fits relative to existing approaches, Section 5 reports quality metrics, and Section 6 draws conclusions and sketches the roadmap ahead.

2. Software Description

2.1. Software Architecture

The system follows a modular, layered client-server architecture (Figure 1). Every analysis stage is independently testable and can be invoked via REST endpoints without using the graphical interface.

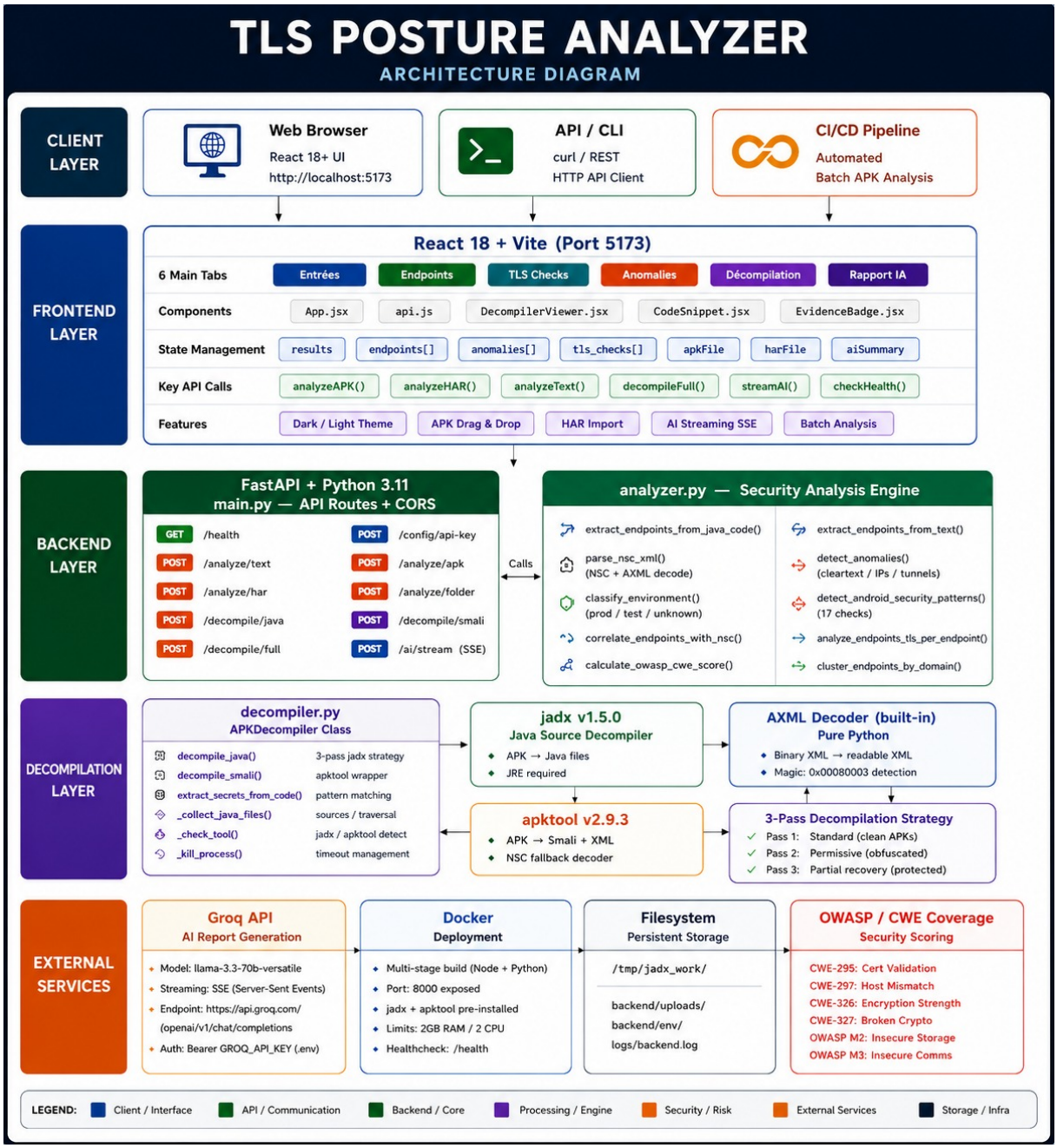

Figure 1: Software architecture of TLS Posture Analyzer. The system is organised into four layers: **Client Layer** (Web Browser via React 18 UI, API/CLI via curl/REST, and CI/CD Pipeline for automated batch analysis); **Frontend Layer** (React 18 + Vite, port 5173, six tabs: Entrées, Endpoints, TLS Checks, Anomalies, Décompilation, Rapport IA); **Backend Layer** (FastAPI + Python 3.11, `main.py` for API routes and `analyzer.py` as the security analysis engine); **Decompilation Layer** (`decompiler.py` with three-pass jadx strategy and apktool fallback, AXML decoder); **External Services** (Groq API for AI report generation, Docker for deployment, filesystem for persistent storage, OWASP/CWE coverage scoring). The dashed boundary marks the trust perimeter between the back-end and the external Groq cloud.

The system is organised into the following layers.

- **Client Layer.** Three entry points: (1) a Web Browser at `http://localhost:5173`, (2) a direct API/CLI interface via `curl`/REST for scripted analysis, and (3) a CI/CD Pipeline connector for automated batch APK analysis.

- **Frontend Layer — React 18 + Vite 5 (port 5173).** A six-tab SPA: *Entrées*, *Endpoints*, *TLS Checks*, *Anomalies*, *Décompilation*, and *Rapport IA*. Key components: `App.jsx`, `api.js`, `DecompilerViewer.jsx`, `CodeSnippet.jsx`, `EvidenceBadge.jsx`. State managed for: `results`, `endpoints[]`, `anomalies[]`, `tls_checks[]`, `apkFile`, `harFile`, `aiSummary`. Key API calls: `analyzeAPK()`, `analyzeHAR()`, `analyzeText()`, `decompileFull()`, `streamAI()`, `checkHealth()`. Supports dark/light theme, APK drag-and-drop, HAR import, AI streaming SSE, and batch analysis.

- **Backend Layer — FastAPI + Python 3.11 (port 8000).** `main.py` exposes REST routes including `GET /health`, `POST /analyze/text`, `POST /analyze/har`, `POST /decompile/java`, `POST /decompile/full`, `POST /config/api-key`, `POST /analyze/apk`, `POST /analyze/folder`, `POST /decompile/smali`, and `POST /ai/stream` (SSE). `analyzer.py` (Security Analysis Engine) implements: `extract_endpoints_from_java_code()`, `parse_nsc_xml()` (NSC + AXML decode), `classify_environment()` (prod/test/unknown), `correlate_endpoints_with_nsc()`, `calculate_owasp_cwe_score()`, `extract_endpoints_from_text()`, `detect_anomalies()` (cleartext, IPs, tunnels), `detect_android_security_patterns()` (17 checks), `analyze_endpoints_tls_per_endpoint()`, and `cluster_endpoints_by_domain()`.

- **Decompilation Layer.** `decompiler.py` (`APKDecompiler` class) with methods `decompile_java()`, `decompile_smali()`, `extract_secrets_from_code()`, `_collect_java_files()`, `_check_tool()`, and `_kill_process()`. Invokes **jadx v1.5.0** (APK→Java, JRE required) and **apktool v2.9.3** (APK→Smali+XML, NSC fallback decoder). The **AXML Decoder** (built-in, pure Python) converts binary XML to readable XML using magic byte `0x00080003` detection.

- **External Services.** Groq API (`llama-3.3-70b-versatile`, streaming via SSE, endpoint `https://api.groq.com/openai/v1/chat/completions`, auth via `GROQ_API_KEY` in `.env`); Docker (multi-stage build, port 8000 exposed, jadx + apktool pre-installed, 2 GB RAM / 2 CPU, healthcheck at `/health`); Filesystem (`/tmp/jadx_work/`, `backend/uploads/`, `backend/env/`, `logs/backend.log`).

Inputs accepted: APK file (.apk), HAR file (.har), raw strings (text), NSC XML paste, proxy logs (Burp Suite).
Outputs produced: a JSON object containing `endpoints[]` (url, classification, tls_risk, nsc_covered), `anomalies[]` (type, value, severity), `findings[]` (pattern, severity, cwe, file, line, snippet), `nsc` (cleartext, pinning, user_ca, min_tls, debug); AI report (Markdown) streamed via SSE; export as JSON/PDF; Docker-ready deployment.

2.2. Analysis Pipeline

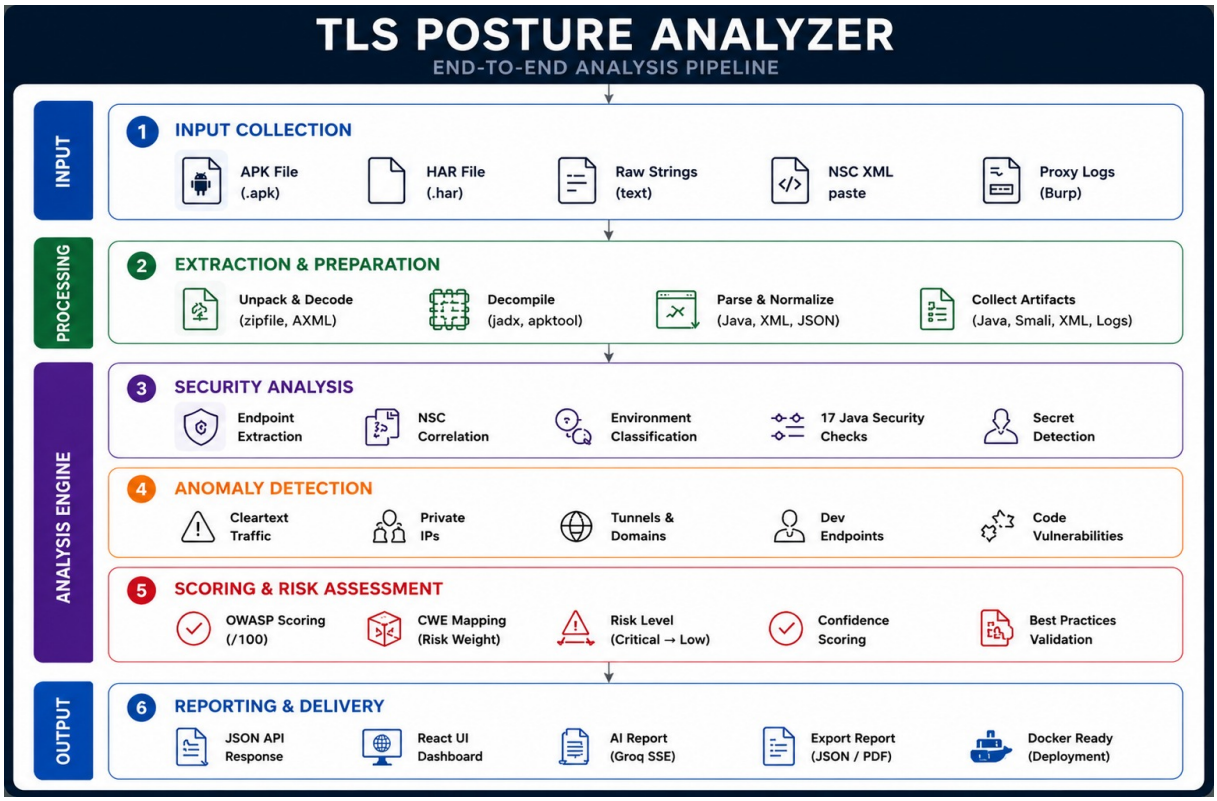

Figure 2: End-to-end analysis pipeline of TLS Posture Analyzer. **Step 1 — Input Collection:** APK file, HAR file, raw strings, NSC XML, proxy logs (Burp). **Step 2 — Extraction & Preparation:** unpack and AXML decode (zipfile), decompile (jadx, apktool), parse and normalise (Java, XML, JSON), collect artefacts (Java, Smali, XML, logs). **Step 3 — Security Analysis:** endpoint extraction, NSC correlation, environment classification, 17 Java security checks, secret detection. **Step 4 — Anomaly Detection:** cleartext traffic, private IPs, tunnels and domains, dev endpoints, code vulnerabilities. **Step 5 — Scoring & Risk Assessment:** OWASP scoring (/100), CWE mapping (risk weight), risk level (Critical→Low), confidence scoring, best-practices validation. **Step 6 — Reporting & Delivery:** JSON API response, React UI dashboard, AI report (Groq SSE), export report (JSON/PDF), Docker-ready deployment.

When an APK is submitted, the backend chains the following steps:

1. **ZIP Extraction.** The APK is opened as a ZIP archive; binary resources (NSC, manifest, `strings.xml`, assets) are extracted to `/tmp/jadx_work/`.

2. **Pure-Python AXML Decoding.** `network_security_config.xml` is reconstructed from compiled Android binary XML using the built-in decoder, identified by magic byte `0x00080003`.

3. **Three-Pass `jadx` Decompilation.** Pass 1: standard mode (clean APKs). Pass 2: permissive mode (obfuscated bytecode). Pass 3: partial-recovery of files produced before failure. `apktool` provides Smali and NSC fallback decoding.

4. **Endpoint Extraction.** Eight Java-specific regex patterns extract URLs, domains, and IPs from string constants, OkHttp calls, Retrofit builders, and `URL`/`URI` constructors.

5. **Environment Classification and NSC Correlation.** Each endpoint is classified as `PROD`, `TEST`, or `UNKNOWN` and correlated with NSC coverage.

6. **Anomaly Detection.** The engine flags cleartext HTTP, private-IP exposure, tunnel indicators, dev endpoints, and code vulnerabilities.

7. **17 Java Security Pattern Detectors.** Each detector runs a primary (presence) regex and a secondary (exploitability) regex, returning `{file, line, snippet}` evidence. OWASP/CWE scores are computed.

8. **Reporting and Delivery.** All six UI tabs are populated from the JSON response. The AI report is generated via Groq SSE and displayed with a typewriter effect. Results can be exported as JSON or PDF.

2.3. Technical Stack

Table 2 maps each component to its exact version, source file, and REST route, providing full traceability between architecture and implementation.

Table 2: Core components — exact versions, source files, and REST routes. Dependency files: `backend/requirements.txt`, `frontend/package-lock.json`. Pinned release: tag `v1.0.0`.

Layer	**Technology**	**Source file**	**REST route**
Frontend	React 18.2 + Vite 5.0 (Node 18+)	`frontend/src/`	port 5173
Backend	FastAPI 0.115, Uvicorn 0.27, Python 3.11	`backend/main.py`	port 8000
Analysis	Custom regex engine (Python 3.11)	`backend/analyzer.py`	`/analyze`
Decompile	jadx 1.5.0, apktool 2.9.3	`backend/decompiler.py`	`/decompile`
AXML	Pure Python (stdlib only)	`backend/decompiler.py`	(internal)
AI Cloud	Groq API — Llama 3.3 70B	`backend/main.py`	`/ai/stream` (SSE)

Technology Rationale.. FastAPI 0.115 was selected for its native SSE support without extra libraries and its automatic OpenAPI documentation at `/docs`. React 18 concurrent rendering enables real-time UI updates during AI token streaming without blocking the interface. The Groq API provides low-latency inference; however, the tool operates fully — without AI reports — when no API key is configured, preserving offline usability.

Privacy and Data Handling.. Code snippets extracted from decompiled APKs are transmitted to the Groq cloud API as part of the AI report prompt. Analysts must obtain explicit authorisation from APK owners before uploading proprietary applications, and should review and redact sensitive content (API keys, tokens, credentials) before enabling the AI report feature. A local LLM backend via Ollama is planned for future releases to eliminate this cloud dependency.

3. Illustrative Example

This section demonstrates TLS Posture Analyzer through a real analysis session using the F-Droid open-source APK (`F-Droid.apk`), a publicly available and freely redistributable application. The screenshots shown are taken directly from the running tool and represent authentic analysis output.

3.1. Background

F-Droid is a well-known open-source Android app repository client. Because its source code is publicly available, it provides a transparent and reproducible reference APK for demonstrating the tool's analysis capabilities without legal concerns. The tool is designed to work on any APK submitted by the analyst; this example serves as a representative walkthrough of all analysis stages.

*3.2. Step 1 — APK Upload and Input Collection (*Entrées *Tab)*

The analyst opens the TLS Posture Analyzer interface at `http://localhost:5173` and uploads `F-Droid.apk` via the *Entrées* tab (Figure 3). The interface confirms the file is loaded (*F-Droid (1).apk chargé*) and immediately extracts the Network Security Configuration XML, which is displayed in the NSC panel on the right. The *Strings APK (Texte Brut)* panel shows raw string artefacts extracted directly from `classes.dex`.

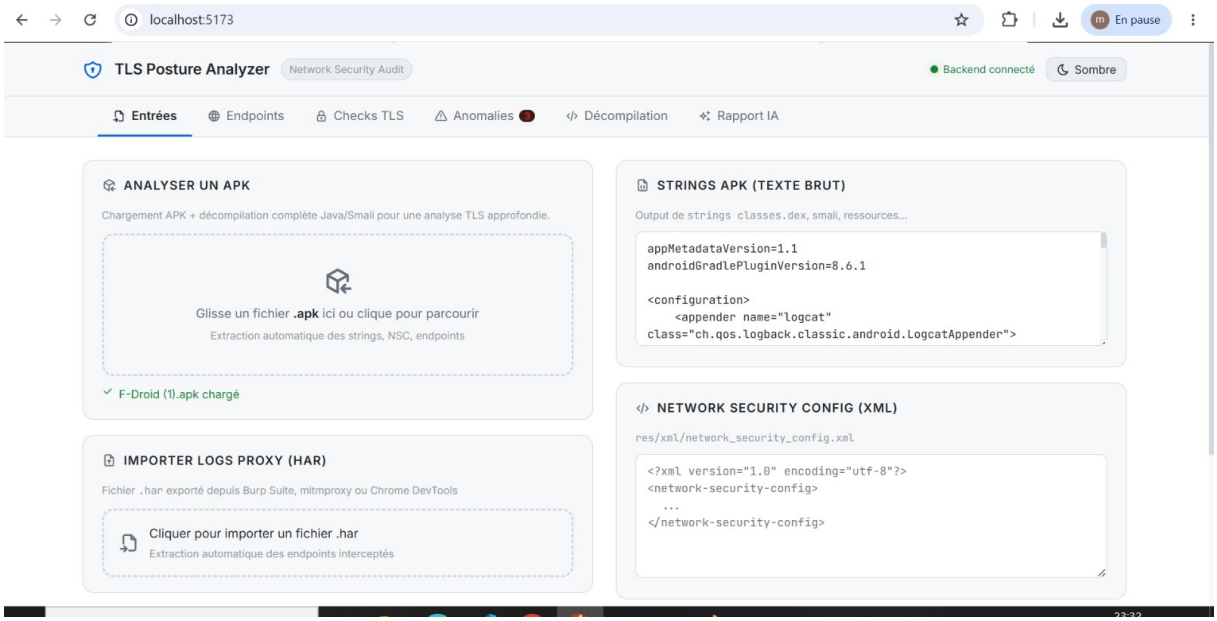

Figure 3: *Entrées* tab after uploading `F-Droid.apk`. Left panel: APK drag-and-drop zone with upload confirmation; HAR proxy log import area below. Right panel: raw strings extracted from `classes.dex` (top) and the decoded `network_security_config.xml` (bottom), automatically parsed by the pure-Python AXML decoder.

*3.3. Step 2 — Endpoint Extraction (*Endpoints *Tab)*

The *Endpoints* tab (Figure 4) lists all URLs, domains, and IP addresses extracted from the decompiled Java source. Each endpoint displays its classification (`PROD`, `TEST`, or `UNKNOWN`), its type (domain or URL), NSC coverage status, and TLS risk level. The tab provides filter buttons (ALL / PROD / TEST / UNKNOWN) and a search field for large APKs. Statistics across TOTAL, PRODUCTION, TEST/STAGING, and HTTP CLAIR categories are shown at the top.

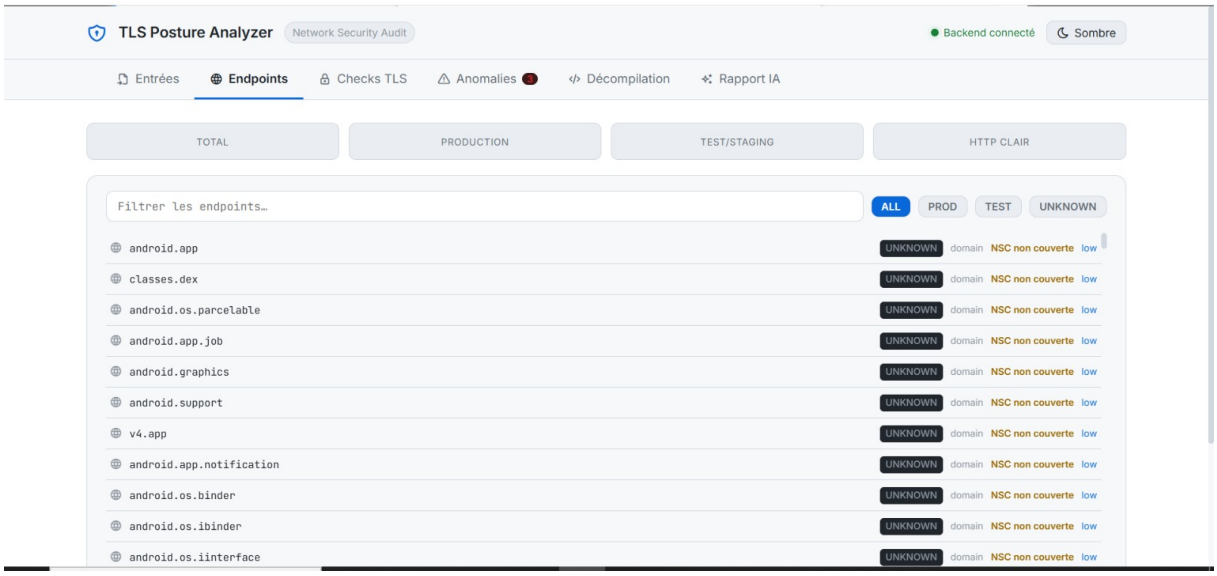

Figure 4: *Endpoints* tab showing extracted domains for `F-Droid.apk`. All detected endpoints are classified as `UNKNOWN` with low TLS risk and flagged as *NSC non couverte* (not covered by a pinning policy). The filter bar allows filtering by environment type. Statistics counters (TOTAL, PRODUCTION, TEST/STAGING, HTTP CLAIR) are displayed at the top of the tab.

*3.4. Step 3 — TLS Security Pattern Detection (*Checks TLS *Tab)*

The *Checks TLS* tab (Figure 5) displays the results of all 17 Java security-pattern detectors plus the NSC verification. The NSC section reports zero findings (no NSC provided via the XML input field). The Java TLS analysis section confirms that 18 controls were analysed; the green banner indicates that decompiled Java code was detected and processed dynamically. Each detector row shows its status badge (*ANALYSÉ*), result, and severity rating (*Faible / Élevé / Critique*). For this APK, the *TrustManager custom*, *HostnameVerifier custom*, and *WebView onReceivedSslError* checks pass with low severity, while *OkHttp CertificatePinner* is flagged as high severity.

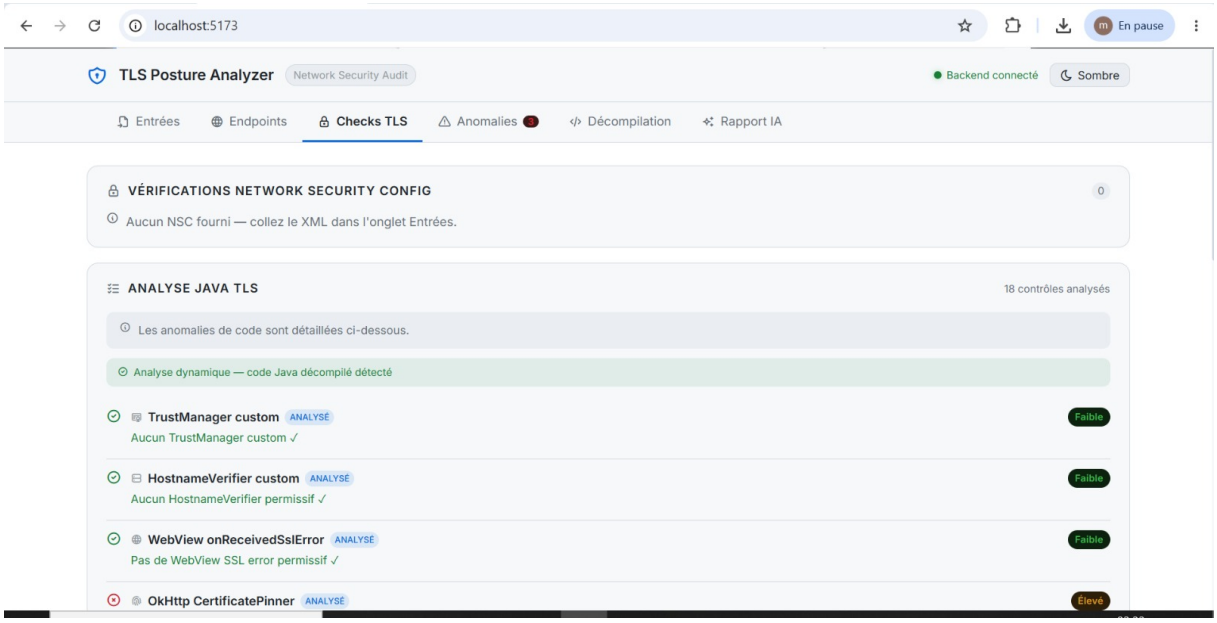

Figure 5: *Checks TLS* tab for `F-Droid.apk`. Top section: NSC verifications (0 findings — no NSC XML pasted). Bottom section: Java TLS analysis across 18 controls. Green banner confirms decompiled Java was successfully detected. Rows show each pattern's status, description, and severity badge (Faible = Low, Élevé = High, Critique = Critical). The *OkHttp CertificatePinner* detector is flagged as High severity.

*3.5. Step 4 — Anomaly Detection (*Anomalies *Tab)*

The *Anomalies* tab (Figure 6) presents a dashboard summarising detected anomalies by severity. For this APK, 3 anomalies are detected in total: 0 CRITICAL, 1 HIGH (*OkHttp CertificatePinner* — no certificate pinning, CA compromise possible), and 2 MEDIUM (*TLS 1.2 minimum* — TLS configuration not explicit, depends on the OS; and *Cipher suite moderne* — cipher suites not specified, uses defaults).

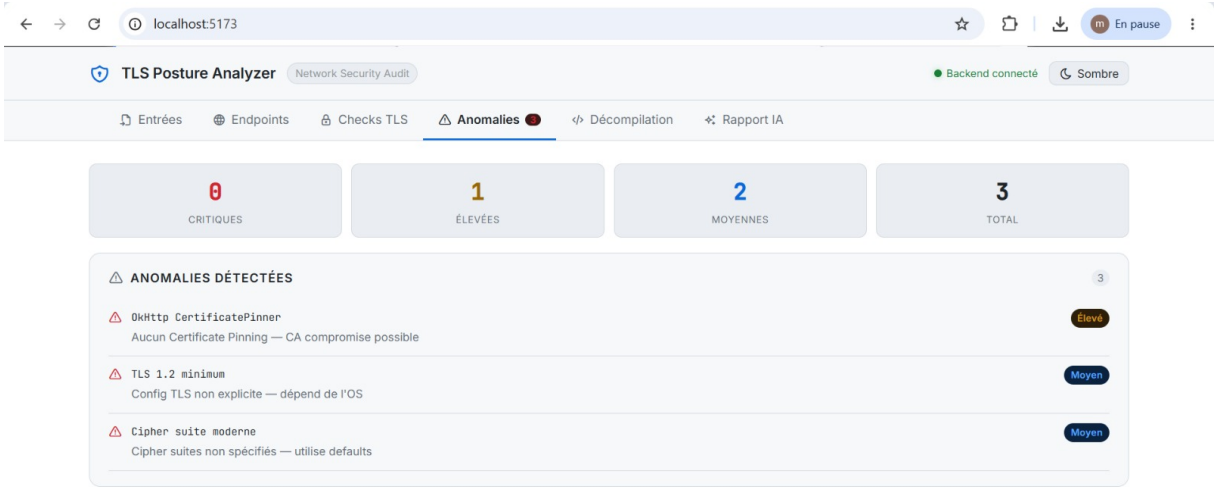

Figure 6: *Anomalies* tab for `F-Droid.apk`. Summary counters: 0 CRITICAL, 1 HIGH, 2 MEDIUM, 3 TOTAL. Detected anomalies listed: OkHttp CertificatePinner (High — no certificate pinning, CA compromise possible), TLS 1.2 minimum (Medium — TLS config not explicit, OS-dependent), Cipher suite moderne (Medium — cipher suites unspecified, using defaults).

*3.6. Step 5 — Decompilation View (*Décompilation *Tab)*

The *Décompilation* tab (Figure 7) provides direct access to the decompiled artefacts. The tool extracted 50 Java files and 100 Smali files from the APK. Three actions are available: *Code Java Complet* (view decompiled Java source), *Assembleur Smali* (view Dalvik bytecode assembly), and *Décompilation Complète* (trigger Static + Smali + Java + secrets analysis). A searchable file explorer lists all decompiled Java files with their package paths and offers individual download buttons.

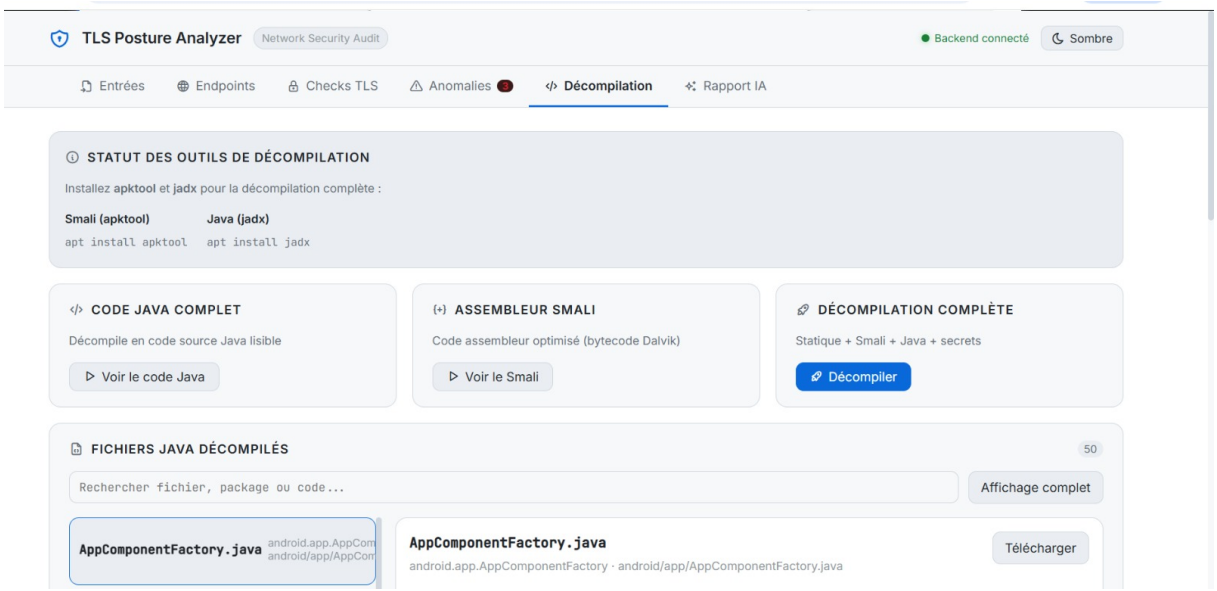

Figure 7: *Décompilation* tab for `F-Droid.apk`. Tool status panel confirms jadx and apktool availability. Three action cards: Java source viewer, Smali assembler, and full decompilation trigger. File explorer lists 50 decompiled Java files; the first selected file (`AppComponentFactory.java`) is shown with its package path and a download button.

3.7. Step 6 — Smali File Browser

Figure 8 shows the Smali file browser, listing 100 Dalvik bytecode assembly files extracted from the APK. Each file entry shows the filename in blue (clickable) and its full path. This view is useful for auditing obfuscated APKs where jadx cannot produce readable Java source, providing an alternative low-level code inspection path.

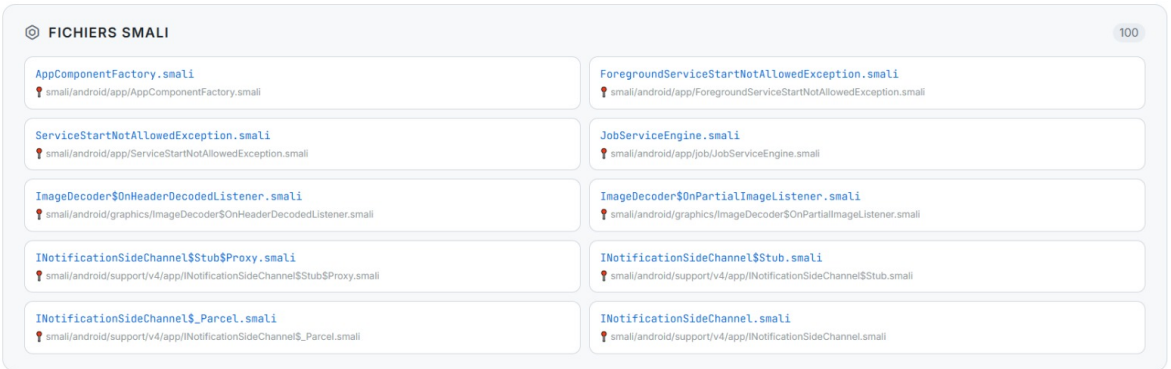

Figure 8: Smali file browser showing 100 Dalvik bytecode assembly files extracted from `F-Droid.apk`. Each card shows the filename and its full path within the APK structure. This view provides fallback inspection capability when jadx decompilation yields no readable Java output.

*3.8. Step 7 — AI-Generated Hardening Report (*Rapport IA *Tab)*

The *Rapport IA* tab (Figure 9) streams the AI analysis in real time when a Groq API key is configured. The structured findings JSON is sent to Llama 3.3 70B via the `/ai/stream` SSE endpoint. The fixed system prompt requests: (1) a risk transport summary with global score, top risks, and environment breakdown, and (2) hardening recommendations including TLS hardening, certificate pinning, HSTS, and a complete NSC example. Generation parameters: `temperature=0.2`, `max_tokens=1024`. The left panel shows the streaming risk summary; the right panel shows the hardening recommendations with concrete NSC XML code samples and a final audit checklist. Recommendations may vary between runs; human review is required before acting on them.

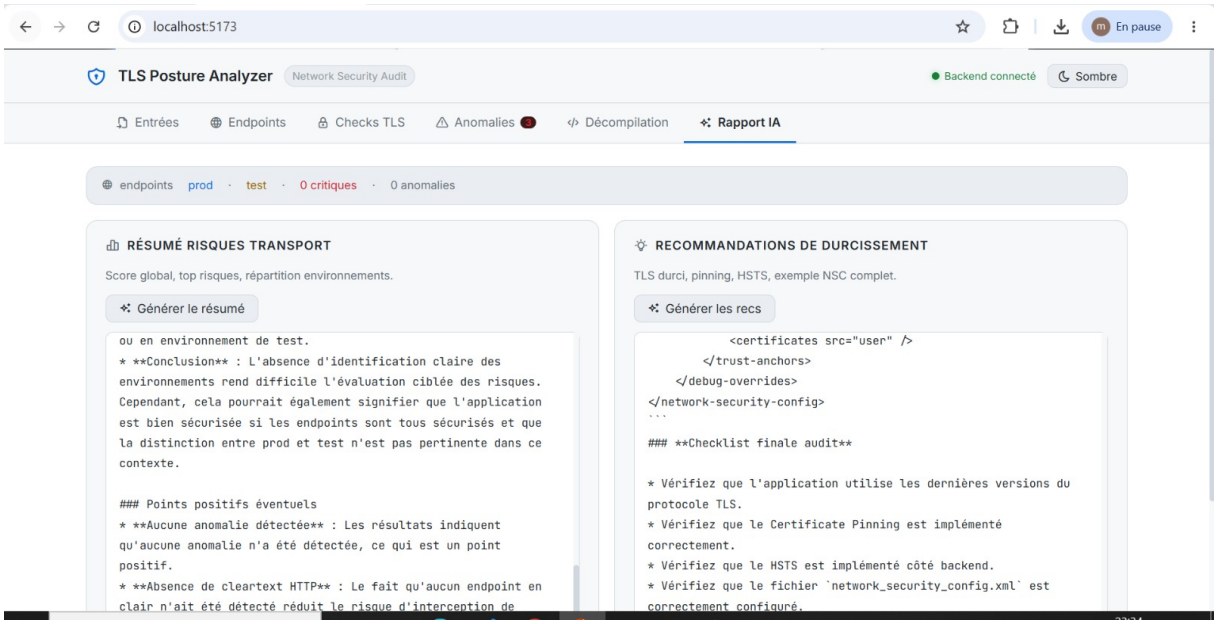

Figure 9: *Rapport IA* tab showing the Llama 3.3 70B analysis streamed in real time for `F-Droid.apk`. Left panel: *Résumé Risques Transport* with score, top risks, and positive findings (0 anomalies, no cleartext HTTP detected). Right panel: *Recommandations de Durcissement* with a complete hardened NSC XML example and a final audit checklist covering TLS version, certificate pinning, HSTS, and NSC file validation.

4. Impact and Discussion

4.1. Security Coverage: 17 Java Pattern Detectors

Each detector in `backend/analyzer.py` implements two compiled `re` patterns applied line-by-line to decompiled Java: (1) a *presence* pattern (high recall, may yield false positives) and (2) an *exploitability* pattern that confirms the anti-pattern is in a reachable, non-debug code path (higher precision). A finding is reported only when the exploitability pattern matches; presence-only matches are retained as lower-confidence warnings. Table 3 documents all 17 detectors.

Table 3: 17 Java TLS security-pattern detectors with CWE mapping, exploitability indicator, and known false-positive conditions.

Pattern	Sev.	CWE	Exploitability indicator	Known FP
Custom TrustManager	CRIT	295	Empty `checkServerTrusted()`	Test-only paths
Custom HostnameVerifier	CRIT	297	`verify()` always `true`	None typical
WebView `onReceivedSslError`	CRIT	295	`handler.proceed()`	None typical
WebView mixed content	CRIT	311	`MIXED_CONTENT_ALWAYS_ALLOW`	None typical
Endpoint verification off	CRIT	297	`endpointVerification=false`	None typical
Cleartext traffic permitted	CRIT	319	`usesCleartextTraffic=true`	Debug builds
OkHttp CertPinner absent	HIGH	295	No SHA-256 pins defined	Intentional OS trust
Custom HttpURLConnection	HIGH	295	`setSSLSocketFactory` calls	Legit custom factory
Custom SSLContext	HIGH	295	Insecure `TrustManager` init	Legit factory only
OkHttp Interceptors	HIGH	312	Bearer token visible in chain	Debug-only logging
Retrofit without pinning	HIGH	295	No `CertificatePinner`	Intentional OS trust
Self-signed certs accepted	HIGH	295	`getAcceptedIssuers()` null	Internal test env
Custom ProxySelector	HIGH	300	`setDefault()` override	Corporate proxy
Custom Authenticator	HIGH	522	`setDefault()` override	Legit credential handler
Minimum TLS version	MED	326	SSLv3 or TLSv1.0 detected	Legacy servers
Missing modern cipher suite	MED	327	No `MODERN_TLS` spec	Older API targets
INTERNET permission	LOW	—	Informational flag only	Not exploitable alone

The "OkHttp CertificatePinner absent" and "Retrofit without pinning" rules flag all usages of these classes that lack explicit pinning. Many legitimate applications intentionally rely on the system certificate store. Absence of pinning is therefore a HIGH-severity recommendation, not a confirmed vulnerability; analysts must cross-reference with the application's threat model.

4.2. Use Cases

Table 4 lists the primary use cases supported by the tool, each justified by the analysis scenarios demonstrated in Section 3.

Table 4: Primary use cases of TLS Posture Analyzer.

Use Case	**Description**
Penetration Testing	Rapid automated TLS posture check before manual deep-dive; seven-step pipeline produces a complete preliminary report in minutes.
Security Engineering	CI/CD integration via the `POST /analyze/folder` batch endpoint to catch TLS regressions before release.
Bug Bounty	Quickly surfaces cleartext HTTP, missing pinning, and NSC misconfiguration with file-and-line evidence.
Compliance Audit	Generates structured JSON evidence for OWASP M2/M3 compliance reports with CWE mapping and OWASP scoring (/100).
Academic Research	Large-scale batch analysis of Android apps from app stores via the REST API without using the graphical interface.

4.3. Comparative Case Study: InsecureBankv2 vs MobSF v4.5

To validate the positioning of *TLS Posture Analyzer* against a reference tool, we submitted the APK `InsecureBankv2 (v1.0)` — a deliberately vulnerable application maintained by Security Innovation — to both tools under identical conditions (local machine, same APK version, same execution date). MobSF v4.5 assigned an overall score of **28/100** (grade F, CRITICAL risk). *TLS Posture Analyzer* focused its analysis exclusively on the transport layer, producing 3 TLS anomalies and an AI-generated remediation summary.

Table 5 summarises the results across five transport layer dimensions.

Table 5: Results comparison on `InsecureBankv2 v1.0`: MobSF v4.5 (report of 23 May 2026) vs *TLS Posture Analyzer* v1.0.0.

Dimension analysed	**MobSF v4.5**	**TLS Posture Analyzer**
v1-only signature (Janus / CVE-2017-13156)	**DETECTED — HIGH**	*Out of scope (signature)*
Cleartext HTTP traffic (`cleartextTraffic`)	No NSC provided — not evaluated	HTTPS required ✓
Certificate Pinning (OkHttp)	Not reported	**DETECTED — High**
Minimum TLS version	Not reported	**MEDIUM**: config not explicit
Modern cipher suites	Not reported	**MEDIUM**: OS defaults
Hardcoded secrets (keys, tokens)	**17 secrets extracted**	*Out of scope (secrets)*
Embedded trackers	3 trackers (Google AdMob, Analytics, GTM)	*Out of scope*
Contextualised remediation report	*Not available*	**Generated by Llama 3.3 70B**
Analysis time	≈ 90 s (decompilation + full SAST)	≈ **45 s** (TLS only)
Required infrastructure	Docker / dedicated server	**Lightweight: Python + Node.js**

Gap analysis. MobSF excels at broad coverage: it detected the Janus vulnerability (v1-only signature), 17 potential hardcoded secrets, and 3 advertising trackers — all outside the TLS scope of our tool. However, MobSF did not flag the absence of Certificate Pinning or the implicit TLS configuration (version and cipher suites), two transport risks identified by *TLS Posture Analyzer* with file-and-line localisation. Furthermore, MobSF provides no ready-to-use NSC remediation; our tool automatically generated a hardened `network_security_config.xml` block together with an OkHttp `CertificatePinner` usage example. This complementarity confirms that *TLS Posture Analyzer* does not replace MobSF but addresses a blind spot: TLS specialisation with AI-driven contextual remediation.

4.4. Comparison with Existing Tools

Table 6 compares TLS Posture Analyzer with MobSF v3.9 [8], QARK v4.0 [9], and Drozer v2.4 [10] on documented features. All three peer tools are actively maintained open-source projects as of the writing of this paper. Installation complexity is measured by the number of mandatory terminal commands required before first APK analysis ($\leq$3: Low; 4–7: Medium; $\geq$8: High).

Table 6: Feature comparison with peer tools. ✓ = full support; △ = partial; × = not supported.

Feature	**This tool**	**MobSF v3.9**	**QARK v4.0**	**Drozer v2.4**
Deep TLS / NSC analysis	✓ Deep	△ Basic	✓	×
Java decompilation (jadx)	✓ 3-pass	✓ jadx	△	×
AI-generated report (LLM)	✓ Groq	×	×	×
HAR / proxy log import	✓	△	×	×
Batch APK analysis	✓	✓	×	×
Built-in AXML decoder	✓	△ apktool	×	×
Cert. pinning (code+NSC)	✓	△ NSC only	✓	×
File+line evidence viewer	✓	×	×	×
OWASP/CWE scoring	✓	✓	△	×
Docker deployment	✓	✓	×	×
Install complexity	Low (2 cmds)	High (≥8)	Medium	High
Open source	✓	✓	✓	✓

Note: this comparison is feature-based. A rigorous experimental comparison on identical APK corpora is deferred to future work.

TLS Posture Analyzer offers deeper TLS-specific analysis, a built-in AXML decoder, file-and-line evidence, and AI-generated remediation within a lower-dependency package than the surveyed peer tools. From an **economic** standpoint, automating the TLS audit workflow reduces preliminary report time from hours to minutes. From an **academic** standpoint, the batch endpoint enables large-scale empirical studies on TLS misconfiguration prevalence. From a **security-engineering** standpoint, the planned CI/CD plugin will shift compliance checks left in the development lifecycle.

5. Quality Assurance

Table 7: Quality metrics for backend and frontend components.

Metric	**Backend**	**Frontend**
Security Rating	A	A
Maintainability Rating	A	B
Reliability Rating	B	A
Open Vulnerabilities	0	0
Open Bugs	0	0
Code Duplication	<2%	<3%
Detector unit tests	17/17	—

The backend returns HTTP 422 on malformed input before any decompilation starts, and HTTP 500 with partial results and a `warning` field on decompilation failure. A `GET /health` endpoint is exposed for liveness checking. Unit tests for all 17 detectors reside in `backend/tests/` and are runnable with:

```
cd backend
pip install -r requirements.txt
pytest tests/ -v
```

Listing 1: Running the unit test suite (expected: 17 passed, 0 failed).

6. Conclusion

This paper presents TLS Posture Analyzer, an open-source platform that automates the complete Android TLS audit workflow. The tool combines a pure-Python AXML decoder, a three-pass `jadx` decompilation strategy, 17 Java security-pattern detectors covering CWE-295, CWE-297, CWE-326, CWE-327, and OWASP M2/M3, and an AI reporting layer backed by Llama 3.3 70B — all accessible through a FastAPI back-end and a React 18 SPA with a two-command installation. The analysis session demonstrated on `F-Droid.apk` confirms that the complete seven-step pipeline — from APK upload through endpoint extraction, TLS pattern detection, anomaly detection, decompilation inspection, and AI-assisted hardening recommendations — produces actionable security intelligence in a fully automated and reproducible manner.

Based on a functional feature comparison, TLS Posture Analyzer provides deeper TLS-specific detection, a built-in AXML decoder, file-and-line evidence, AI-generated remediation, and lower installation complexity than the surveyed peer tools; a rigorous quantitative comparison on real-world APK corpora remains future work.

Current Limitations and Threats to Validity

- **Static-analysis threat.** Runtime TLS handshakes are not observed. Pairing with Burp Suite or `mitmproxy` is recommended for full coverage.
- **Obfuscation threat.** Heavily packed APKs (DexGuard, AppShielding) may yield zero decompiled Java files; the three-pass strategy recovers partial output but cannot defeat commercial bytecode encryption.
- **AXML coverage threat.** Rare chunk types may produce incomplete XML, triggering a regex fallback.
- **LLM dependency threat.** AI reports require an active Groq API key and internet connectivity; free-tier rate limits may throttle batch workflows. Recommendations are non-deterministic.
- **No persistence.** The tool is designed for a single analyst session; no authentication or server-side storage is implemented.

Roadmap

The following features are planned for future releases and are *not* available in v1.0.0:

1. **Dynamic TLS probe** — Frida hooks or `mitmproxy` subprocess to verify runtime TLS handshakes against static findings.
2. **Database persistence** — SQLite/PostgreSQL backend to store, diff, and trend analyses across APK versions.

3. **CI/CD plugin** — GitHub Actions / GitLab CI action that fails the pipeline when findings exceed a threshold, with SARIF export [13] for GitHub Advanced Security.

4. **Kotlin / Flutter support** — Extension of the pattern engine to Kotlin and Flutter/-Dart.

5. **Local LLM (Ollama)** — Local inference backend to eliminate the Groq cloud dependency and enable fully offline use.

6. **Formal evaluation** — Precision/recall/F1 benchmark on a labelled real-world APK corpus with human-annotated ground truth and analysis-time measurements.

Ethical and Legal Considerations

TLS Posture Analyzer is intended solely for the analysis of APKs for which the analyst holds explicit authorisation: own applications, published bug-bounty programmes, or applications provided by the owner for security assessment. Analysing third-party proprietary applications without permission may violate applicable computer-fraud and reverse-engineering laws. Extracted secrets identified in decompiled source code must be handled according to responsible disclosure practices. When the AI report feature is enabled, code snippets — potentially containing secrets — are transmitted to the Groq cloud API; analysts should review and redact sensitive content before enabling this feature on production APKs.

Availability and Reuse

- **Repository:** `https://github.com/M4l1k40/TLS-Posture-Analyzer`
- **Release tag:** `v1.0.0`
- **Licence:** MIT
- **Documentation:** `https://github.com/M4l1k40/TLS-Posture-Analyzer/blob/main/README.md`
- **Support e-mail:** `lachgar.m@gmail.com`

Reproducibility Checklist

1. **Environment:** OS Linux/macOS/Windows; Python 3.11+; Node.js 18+; jadx 1.5.0; Docker (optional).

2. **Commands:**

```
git clone https://github.com/M4l1k40/TLS-Posture-Analyzer
cd TLS-Posture-Analyzer
# Option A: native
chmod +x start.sh && ./start.sh    # Linux/macOS
start.bat                           # Windows
# Option B: Docker
docker compose up --build
```

3. **Expected output:** Upload any APK; the tool runs seven analysis stages and populates all six tabs. Upload `F-Droid.apk` to reproduce the results shown in Figures 3–9.

4. **Known limitations:** AI report requires a valid Groq API key; without it, the *Rapport IA* tab shows a configuration warning. Obfuscated APKs may yield fewer Java files than expected.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We would like to express our sincere gratitude to our supervisor, Professor Lachgar, for his valuable guidance, availability, and support throughout this project. We also extend our thanks to the faculty members of the École Nationale des Sciences Appliquées de Marrakech for the quality of education provided within the Cyber Defense and Embedded Telecommunications Systems program. Finally, we would like to express our appreciation to everyone who contributed, directly or indirectly, to the successful completion of this work.

References

[1] Android Developers. Network Security Configuration. `https://developer.android.com/privacy-and-security/network-security-config`, 2023.

[2] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, M. Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in)security. In *Proc. ACM CCS*, pp. 50–61, 2012.

[3] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, S. Fahl. To pin or not to pin — Helping app developers bullet proof their TLS connections. In *Proc. USENIX Security Symposium*, pp. 239–254, 2015.

[4] OWASP Foundation. OWASP Mobile Top 10: M2 Insecure Data Storage, M3 Insecure Communication. `https://owasp.org/www-project-mobile-top-10/`, 2016.

[5] OWASP Foundation. Mobile Application Security Verification Standard (MASVS) v2.0. `https://mas.owasp.org/MASVS/`, 2023.

[6] OWASP Foundation. Mobile Application Security Testing Guide (MASTG). `https://mas.owasp.org/MASTG/`, 2023.

[7] MITRE Corporation. Common Weakness Enumeration: CWE-295, CWE-297, CWE-326, CWE-327. `https://cwe.mitre.org/`, 2024.

[8] A. Abraham. MobSF: Mobile Security Framework v3.9. `https://github.com/MobSF/Mobile-Security-Framework-MobSF`, 2024.

[9] LinkedIn Corporation. QARK: Quick Android Review Kit v4.0. `https://github.com/linkedin/qark`, 2019.

[10] WithSecure Labs. Drozer v2.4. https://github.com/WithSecureLabs/drozer, 2023.

[11] skylot. jadx: Dex to Java decompiler v1.5.0. https://github.com/skylot/jadx, 2024.

[12] C. Tumbleson, R. Wiśniewski. Apktool v2.9.3 — A tool for reverse engineering Android APK files. https://apktool.org/, 2023.

[13] OASIS Open. Static Analysis Results Interchange Format (SARIF) v2.1. https://docs.oasis-open.org/sarif/sarif/v2.1.0/, 2019.

[14] Groq Inc. Groq API documentation — Llama 3.3 70B. https://console.groq.com/docs/, 2024.

[15] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén. *Experimentation in Software Engineering*. Springer, 2012.