

TV Screen DSP — Technical Report

Project: TV Screen Crack Detection via DSP

Package: com.example.tvscreendsp

Date: February 12, 2026

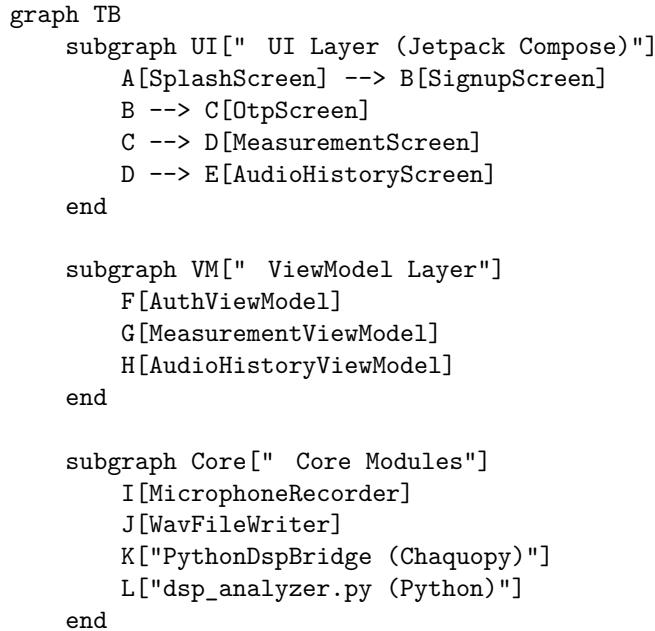
Platform: Android (minSdk 26, targetSdk 36)

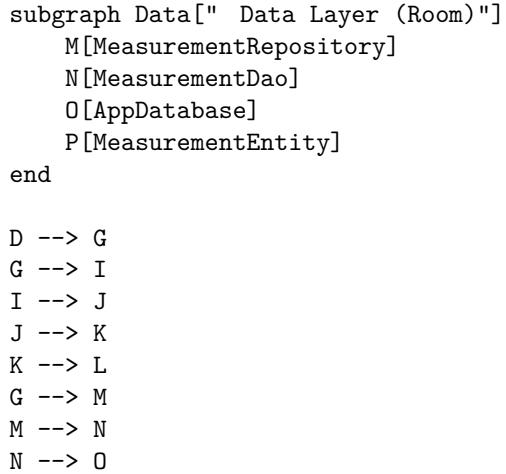
1. Executive Summary

TV Screen DSP is a native Android application that uses **Digital Signal Processing (DSP)** to detect cracks in TV screens through audio analysis. The app records audio from the device microphone, processes the WAV file through a **Python DSP engine embedded inside the Android app** (via Chaqueony), and classifies the signal as **CRACK**, **NORMAL**, or **NOISE** with a confidence score.

The architecture employs a **Kotlin Python interop** pattern where Kotlin handles the Android lifecycle, UI (Jetpack Compose), and data persistence (Room), while Python handles all DSP computations using only the standard library — no NumPy or SciPy dependencies.

2. Project Architecture Overview





3. Package Structure & File Inventory

Package	File	Lines	Purpose
audio	AudioConfig.kt	36	Recording constants (44.1kHz, 16-bit, mono)
	MicrophoneRecorder.kt	218	Android AudioRecord wrapper, 10s capture
	WavFileWriter.kt	109	PCM → WAV file conversion (44-byte RIFF header)
	RecordingState.kt	21	Sealed class: Idle → Recording → Completed → Error
	AudioRecordPitfalls.kt	153	Developer reference doc (7 pitfall categories)
dsp	PythonDspBridge.kt	166	Kotlin - Python bridge via Chaquopy
	ChaquopyPitfalls.kt	197	Developer reference doc (9 pitfall categories)

Package	File	Lines	Purpose
python	dsp_analyzer.py	212	Pure Python DSP engine (no external deps)
data.model	DspResult.kt	46	Data class for DSP output
data.local	AppDatabase.kt	101	Room DB (singleton, version 2)
	MeasurementEntity.kt	105	Room entity with nullable DSP result fields
	MeasurementDao.kt	20	DAO with Flow-based reactive queries
	RoomPitfalls.kt	—	Developer reference doc
data.repository	MeasurementRepository.kt	149	Repository pattern abstraction
auth	AuthManager.kt	62	SharedPreferences-based demo auth
ui.splash	SplashScreen.kt	188	Animated splash with dark gradient
ui.auth	SignupScreen.kt	—	Phone/email input screen
	OtpScreen.kt	—	6-digit OTP verification
	AuthViewModel.kt	126	Input validation, local OTP generation
ui.measurement	MeasurementScreen.kt	366	Main screen: record + analyze + results
	MeasurementViewModel.kt	269	Orchestrates recording → DB → DSP pipeline
ui.history	AudioHistoryScreen.kt	342	LazyColumn list with playback, rename, delete
	AudioHistoryViewModel.kt	—	History data management
root	MainActivity.kt	127	NavHost, Python bridge init, 5 routes

Total: ~40+ source files across 7 Kotlin packages + 1 Python module

4. How Kotlin Interacts with Python (Chaquopy Bridge)

This is the **most critical architectural decision** in the project. The app embeds a Python 3.8 runtime inside the Android APK using Chaquopy.

4.1 Architecture Diagram

```
sequenceDiagram
    participant UI as MeasurementScreen
    participant VM as MeasurementViewModel
    participant Bridge as PythonDspBridge (Kotlin)
    participant Py as dsp_analyzer.py (Python)
    participant DB as Room Database

    UI->>VM: startRecording()
    Note over VM: Records 10s audio → WAV file
    VM->>DB: createMeasurement(wavPath)
    DB-->>VM: measurementId
    VM->>Bridge: analyzeAudio(wavPath)
    Bridge-->>Bridge: analysisMutex.withLock { }
    Bridge->>Py: callAttr("analyze_audio", wavPath)
    Note over Py: read_wav() → calculate_power_db()<br/>→ find_dominant_frequency()<br/>→ ca
    Py-->>Bridge: Python dict {frequency, power, ...}
    Bridge->>Bridge: convertPyObjectToResult(pyDict)
    Bridge-->>VM: DspResult (Kotlin data class)
    VM->>DB: updateWithDspResults(id, dspResult)
    VM-->>UI: AnalysisState.Completed(result)
```

4.2 Initialization (Main Thread Only)

```
// In MainActivity.onCreate() - MUST be main thread
PythonDspBridge.initialize(this)

// Internally:
fun initialize(context: Context) {
    if (!Python.isStarted()) {
        Python.start(AndroidPlatform(context)) // Boots Python runtime
    }
    dspModule = Python.getInstance().getModule("dsp_analyzer") // Loads .py
}

[!IMPORTANT] Python initialization must happen on the main thread. Calling Python.start() from a background thread causes native crashes.
```

4.3 Cross-Language Call (Background Thread)

```
// Runs on Dispatchers.Default (CPU thread pool)
suspend fun analyzeAudio(wavPath: String): DspResult? = withContext(Dispatchers.Default) {
    analysisMutex.withLock { // Only one Python call at a time (GIL safety)
        val pyResult = dspModule!!.callAttr("analyze_audio", wavPath)
        convertPyObjectToResult(pyResult) // Python dict → Kotlin data class
    }
}
```

4.4 Data Type Conversion (Python → Kotlin)

Python Type	Bridge Method	Kotlin Type
dict	pyDict.asMap()	Map<PyObject, PyObject>
float	value.toDouble()	Double
str	value.toString()	String

[!WARNING] Using `pyDict.get("key")` calls Python's `getattr()`, NOT `dict[key]`. The bridge correctly uses `asMap()` to iterate over dictionary entries.

4.5 Build Configuration (Chaquopy)

```
// settings.gradle.kts - Maven repository
maven { url = uri("https://chaquo.com/maven") }

// app/build.gradle.kts
plugins { alias(libs.plugins.chaquopy) }

chaquopy {
    defaultConfig {
        version = "3.8"
        // No pip packages - pure Python stdlib only
    }
}

// ABI filtering (reduces APK size)
ndk { abiFilters += listOf("arm64-v8a", "x86_64") }
```

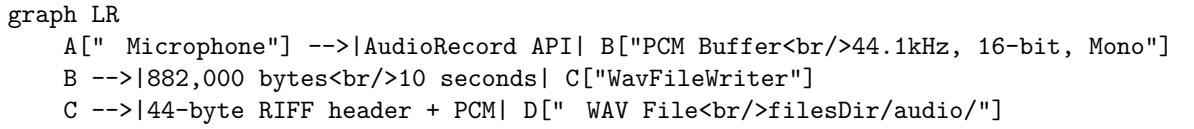
4.6 Thread Safety Model

Concern	Solution
Python GIL conflicts	Kotlin Mutex serializes all calls
UI blocking during DSP	Dispatchers.Default for CPU work

Concern	Solution
Python init thread	<code>synchronized(this) + double-checked lock</code>
Memory leaks	PyObject references scoped to function calls

5. How DSP Works in the App

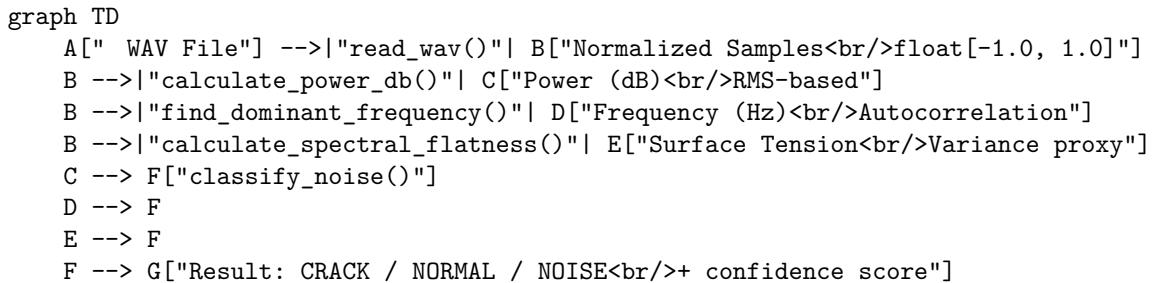
5.1 Audio Capture Pipeline



Recording parameters: - **Sample rate:** 44,100 Hz (CD quality, suitable for DSP) - **Bit depth:** 16-bit PCM (adequate dynamic range: ~96 dB) - **Channels:** Mono (single microphone source) - **Duration:** Exactly 10 seconds - **Total data:** $44,100 \times 10 \times 2 \text{ bytes} = 882,000 \text{ bytes}$ per recording - **Storage:** Internal app storage (`filesDir/audio/measurement_YYYYMMDD_HH:mm:ss.wav`)

5.2 DSP Analysis Pipeline (Python)

The DSP engine (`dsp_analyzer.py`) processes the WAV file through 4 stages:



Stage 1 — WAV File Reading (`read_wav`)

- Opens file using Python's `wave` module (`stdlib`)
- Reads 16-bit PCM samples via `struct.unpack()`
- Converts stereo to mono if needed
- Normalizes to $[-1.0, 1.0]$ range by dividing by 32,768

Stage 2 — Power Calculation (`calculate_power_db`)

$$\text{RMS} = \sqrt{\sum(\text{sample}^2) / N}$$

$$\text{Power_dB} = 20 \times \log (\text{RMS})$$

- Threshold: signals below **-50 dB** are classified as **NOISE** immediately

- Uses the standard decibel formula for signal strength measurement

Stage 3 — Frequency Detection (`find_dominant_frequency`)

- Uses **autocorrelation** (not FFT) — pure Python, no NumPy needed
- Calculates correlation between signal and time-shifted copies of itself
- Searches lags from 20 to 2000 samples (first 8192 samples for speed)
- The lag with maximum correlation → `frequency = sample_rate / best_lag`
- This detects the dominant periodic component in the signal

Stage 4 — Spectral Flatness / Surface Tension (`calculate_spectral_flatness`)

- Computes variance of the sample values as a proxy for spectral spread
- Normalizes to [0, 1] range: `flatness = min(1.0, sqrt(variance) * 10)`
- Higher variance = more noise-like signal = higher flatness score
- This measures how “spread out” the frequency content is

Stage 5 — Classification (`classify_noise`)

Rule-based classification using three crack indicators:

Indicator	Threshold	Reasoning
High frequency	> 1,500 Hz	Cracks produce sharp, transient sounds
High spectral flatness	> 0.6	Crack noise has broad frequency content
Strong signal power	> -20 dB	Cracks are typically loud events

Decision logic: - 2+ indicators → **CRACK** (confidence: 0.5 + indicators × 0.2, max 0.9) - 1 indicator → **NORMAL** (confidence: 0.7) - 0 indicators → **NORMAL** (confidence: 0.8) - Power < -40 dB → **NOISE** (confidence: 0.6)

5.3 DSP Result Data Model

```
data class DspResult(
    val frequency: Double,           // Dominant frequency in Hz
    val power: Double,               // Signal power in dB (RMS)
    val surfaceTension: Double,      // Spectral flatness [0-1]
    val noiseStatus: String,         // "CRACK" | "NORMAL" | "NOISE"
    val confidence: Double          // Classification confidence [0-1]
)
```

6. Data Persistence Layer

6.1 Database Schema (Room, Version 2)

Column	Type	Nullable	Description
<code>id</code>	Long	No	Auto-generated PK
<code>wavFilePath</code>	String	No	Path to WAV file
<code>recordedAt</code>	Long	No	Unix timestamp (ms)
<code>customName</code>	String	Yes	User-assigned label (added in v2 migration)
<code>inputSource</code>	String	No	“MICROPHONE”, “USB”, or “BLE”
<code>frequency</code>	Double	Yes	Populated after DSP analysis
<code>power</code>	Double	Yes	Populated after DSP analysis
<code>surfaceTension</code>	Double	Yes	Populated after DSP analysis
<code>noiseStatus</code>	String	Yes	Populated after DSP analysis
<code>confidence</code>	Double	Yes	Populated after DSP analysis
<code>analysisCompleted</code>	Boolean	Yes	Populated after DSP analysis

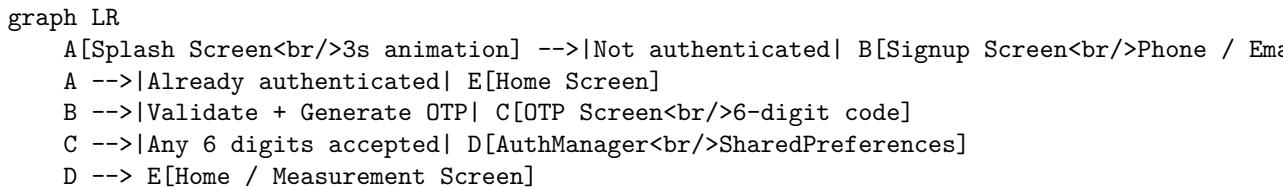
6.2 Repository Pattern

The `MeasurementRepository` provides:

- `createMeasurement()` — insert record immediately after WAV saved (DSP fields null)
- `updateWithDspResults()` — populate DSP fields after analysis completes
- `deleteMeasurement()` — removes both DB record **and** WAV file
- `renameMeasurement()` — user-assigned custom name
- `getAllMeasurements()` — reactive Flow for UI auto-updates

7. Authentication Flow (Demo)

The authentication is a **local-only demo flow** — no backend, no network calls.



- **OTP is generated locally** and logged to Logcat (for demo purposes)

- Any 6-digit code is accepted as valid
 - Authentication state persisted in Shared Preferences
 - Splash screen checks AuthManager.isLoggedIn on app launch
-

8. Navigation Architecture

5 routes managed by Jetpack Navigation Compose:

Route	Screen	Description
splash	SplashScreen	3s animated splash → auto-navigate
signup	SignupScreen	Phone/email input with validation
otp	OtpScreen	6-digit OTP verification
home	MeasurementScreen	Record + analyze (main feature)
history	AudioHistoryScreen	Browse past recordings with playback

9. Complete End-to-End Data Flow

User taps "Measure Noise"

```
MeasurementViewModel.startRecording()
```

```
MicrophoneRecorder.startRecording() → Flow<RecordingState>
(10 seconds of audio capture at 44.1kHz)
(emits progress updates: 0% → 100%)
```

```
WavFileWriter.writePcmToWav()
(44-byte RIFF header + 882,000 bytes PCM data)
```

```
MeasurementRepository.createMeasurement()
(Room insert, DSP fields = null)
```

```
PythonDspBridge.analyzeAudio(wavPath)
```

```
PYTHON RUNTIME (Chaquopy)
dsp_analyzer.analyze_audio()
1. read_wav → normalized floats
```

```

    2. calculate_power_db → dB
    3. find_dominant_frequency → Hz
    4. calculate_spectral_flatness
    5. classify_noise → status

convertPyObjectToResult() → DspResult

MeasurementRepository.updateWithDspResults()
    (Room update: populate DSP fields + analysisCompletedAt)

UI updates: ResultCard shows Status, Confidence, Frequency, Power, Surface Tension

```

10. Technology Stack Summary

Layer	Technology	Version
Language (Android)	Kotlin	—
Language (DSP)	Python	3.8
Kotlin-Python Bridge	Chaquopy	Via Gradle plugin
UI Framework	Jetpack Compose + Material 3	—
Database	Room (SQLite)	—
Navigation	Jetpack Navigation Compose	—
Permissions	Acccompanist Permissions	—
Build System	Gradle (KTS)	—
Compile SDK	Android 36	—
Min SDK	Android 26 (Oreo)	—
Symbol Processing	KSP (for Room)	—
Java Target	JVM 17	—

11. Current Project Status

Completed Features

- ☒ Audio recording pipeline (10s microphone capture → WAV)
- ☒ Python DSP engine with pure stdlib (no NumPy/SciPy deps)
- ☒ Chaqueopy Kotlin-Python bridge with thread safety
- ☒ Room database with measurement history
- ☒ Results UI showing frequency, power, surface tension, status, confidence
- ☒ Audio history screen with playback, rename, and delete
- ☒ Splash screen with professional animations

- Demo authentication flow (phone/email + OTP)
- Navigation architecture (5 screens)
- Runtime permission handling (RECORD_AUDIO)
- DB migration v1 → v2 (customName column)
- Developer reference docs (AudioRecordPitfalls, ChaquopyPitfalls, Room-Pitfalls)

Planned / Future

- USB audio input support
 - BLE audio input support (entity supports it, not yet implemented)
 - Real backend authentication (currently demo-only)
 - ML-based crack classification (currently rule-based)
 - NumPy integration for FFT-based frequency detection (currently using autocorrelation)
-

12. Key Design Decisions & Rationale

Decision	Rationale
Pure Python stdlib for DSP (no NumPy)	Avoids 10+ min build times and APK size explosion
Autocorrelation instead of FFT	No NumPy dependency; sufficient for dominant frequency detection
Chaquopy for Python embedding	Mature, maintained bridge; Gradle plugin integration
Mutex for Python calls	GIL + Chaquopy interop safety; prevents race conditions
Post-analysis update pattern	WAV is saved immediately; DSP results populate later (fault-tolerant)
Sealed classes for state	Type-safe state machines for recording + analysis states
Flow-based DAO queries	Reactive UI updates without manual refresh
ABI filtering (arm64-v8a + x86_64 only)	Reduces build time from ~30 min to ~10 min