Feature

- Secure Capture — Real-Time, On-Device, No Storage
- Secure Call Verification — Verify Bank-Initiated Calls
- SIM Change Detection — Android Background Service

Secure Capture — Real-Time, On-Device, No Storage

Purpose
Analyze frames from the front-facing camera entirely **on device** to enhance fraud detection—**no images are stored or transmitted; only metadata/results are sent**.

How it works

- App decides when to activate (e.g., after login, during a transfer, or only in high-risk flows).
- `startCapture()` checks camera permission and begins periodic frame analysis locally.
- Frames are processed on device; images are **deleted immediately** after analysis.
- Only analysis results/metadata are sent to your backend.
- Capture ends when the app calls stopCapture().

Required permission (AndroidManifest.xml)

```
<uses-permission android:name="android.permission.CAMERA"/>
```

SDK methods & returns

- `startCapture()` → `SUCCESS` | `MISSING_PERMISSIONS`
- `stopCapture()` → `SUCCESS`

Usage example (Kotlin)

```
val result = collectorAgent.startCapture()
if (result == "MISSING_PERMISSIONS") {
    // Request CAMERA permission before retrying
}
// ... perform secured flow ...
collectorAgent.stopCapture()
```

UI guidance (pre-permission message)}

> “To enhance security, the app will analyze images from your device’s front-facing camera. No images are stored or uploaded.”

Security & privacy notes

- No images ever leave the device; only minimal metadata/results are transmitted.
- Immediate buffer deletion after analysis; follow data minimization best practices.

Common scenarios

- Post-login environment validation
- Extra check during standard transfers
- High-risk events (new payee, unusually large amount)

Next steps

1. Add CAMERA permission.
2. Explain purpose to users.
3. Integrate startCapture()/stopCapture() in the chosen flow(s).
4. Handle permission gracefully. 5) Roll out gradually and monitor analytics.

Secure Call Verification — Verify Bank-Initiated Calls

Purpose
Let the app confirm whether an incoming/recent call was genuinely initiated by the bank—helping stop voice phishing and number spoofing.

System architecture

1. **Bank Call Center** reports each outbound call to the Secure Call backend.
2. **Secure Call Backend** records the call data.
3. Mobile App + SDK queries the backend and displays verification status.

Server-to-server API (ReportCall)
Endpoint: `POST /api/v1/ReportCall`
Fields: `phone_number` (req), `uid` (req), `call_reason` (opt), `call_team` (opt), `call_agent` (opt)

Example (curl)

```
curl -X POST "https://securecall.example.com/api/v1/ReportCall" \
  -H "Authorization: Bearer <your_token>" \
  -H "Content-Type: application/json" \
  -d '{
    "phone_number": "+15551234567",
    "uid": "123456789",
    "call_reason": "Verify Transaction",
    "call_team": "Fraud Department",
    "call_agent": "John Smith"
  }'
```

Android SDK integration
Permission (AndroidManifest.xml):

```
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
```

If missing, `CheckCallStatus()` returns `UNKNOWN`.

SDK method

```
val result = collectorAgent.CheckCallStatus()
```

Statuses

- `CALL_APPROVED` — Legitimate bank-initiated call (active or recent)
- `UNAPPROVED_RECENT_CALL` — Recent call detected but not reported → potential fraud
- `NO_RECENT_CALL` — No active/recent call
- **UNKNOWN** — Insufficient info (incompatibility or missing permission)

Sample returns

Legitimate:

```
{
  "STATUS": "CALL_APPROVED",
  "CALL_REASON": "Verify Transaction",
  "CALL_TEAM": "Fraud Department",
  "CALL_AGENT": "John Smith"
}
```

Unapproved:

```
{ "STATUS": "UNAPPROVED_RECENT_CALL" }
```

No call:

```
{ "STATUS": "NO_RECENT_CALL" }
```

UI recommendations

- `CALL_APPROVED` → Green banner: “This call is verified.” (show agent + reason)
- `UNAPPROVED_RECENT_CALL` → Yellow warning: “Be cautious—possible fraud.”
- **UNKNOWN** → Gray info: “Unable to verify call status.”

End-to-end example

1. Agent calls customer and backend sends `ReportCall`.
2. Customer opens the app during the call.
3. App invokes `CheckCallStatus()` and receives `CALL_APPROVED`.
4. App shows green banner with agent/team/reason.

SIM Change Detection — Android Background Service

Continuously monitor SIM state changes to help mitigate SIM-swap account takeovers. Runs silently in the background and sends only non-sensitive metadata for risk scoring.

High-level flow

1. User installs/opens the banking app with the SDK.
2. A lightweight **background service** starts automatically (and after reboots).
3. Service periodically checks SIM state allowed by Android.
4. If the current SIM differs from the baseline, an event is triggered.
5. Event (NO_CHANGE | SIM_CHANGED | UNKNOWN) + session metadata is sent securely to the backend.

Required permission (AndroidManifest.xml)

```
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
```

Notes: Android 10+ restricts some identifiers; behavior varies by OEM.

Event types

- `NO_CHANGE` — SIM unchanged
- `SIM_CHANGED` — SIM differs from baseline
- `UNKNOWN` — SIM info unavailable on device/OS

Example payload to backend

```
{
  "device_id": "abcd-1234",
  "timestamp": "2025-09-16T15:00:00Z",
  "sim_status": "SIM_CHANGED",
  "carrier": "CarrierName",
  "os_version": "Android 14",
  "sdk_version": "2.1.0"
}
```

Security & privacy

- No SMS/contacts/phone numbers are read; only SIM status + technical metadata.
- TLS 1.2+ for transport; designed for data minimization.

Limitations & compatibility

- Android-only; newer Android versions/OEMs may mask info.
- Dual-SIM devices may complicate detection; not available on iOS.
- Maintain a tested device/OS compatibility list and validate before rollout.

Next steps

1. Add permission.
2. 2) Initialize background service at app start.
3. 3) Ingest events server-side.
4. 4) Combine with other signals for scoring.
5. 5) Validate on your device matrix pre-production.