

System Security Project: MQTTImage

Ioana-Denisa Popescu

George-Andrei Cordis

Faculty of Automatic Control and Computers

University POLITEHNICA of Bucharest

ioana.popescu1905@stud.acs.upb.ro george.cordis@stud.acs.upb.ro

May 26, 2025

1 Introduction

1.1 Context and Problem

It is a commonly held view that, in today's technological landscape, mobile applications play a central role in capturing, storing and organizing images. With the introduction of high-resolution cameras on mobile devices, users often find themselves accumulating numerous similar or identical images in their camera roll¹.

However, despite the significant increase in the amount of personal visual data, users typically do not give sufficient consideration to security and privacy concerns when capturing, storing, or sharing these images. This lack of awareness can lead to substantial vulnerabilities regarding the protection of sensitive information contained within photographs.

This project aims to develop an end-to-end distributed infrastructure consisting of a mobile application that enables image capture via the device's camera, which are further securely transmitted to a web-based backend server that provides various features, where they are processed and stored. The system facilitates remote monitoring, configurable image acquisition and automated image analysis, with strong considerations for security via encrypted data transmission using MQTT over mutual TLS (mTLS)², secure authentication and authorization mechanisms, encrypted storage of images and metadata and continuous monitoring with anomaly detection to identify and mitigate potential threats.

1.2 Proposed Solution and Objectives

To enhance reliability and operational efficiency, the system is built with features such as:

- Image capture and transmission from mobile devices.
- Secure MQTT communication using mutual TLS.
- Multi-mode camera operation and multiple features within the Android application.

¹<https://blog.avast.com/how-many-of-the-photos-that-we-store-are-bad>

²<https://medium.com/@kajsuaning/mqtt-mutual-certificate-authentication-f51bc6e1a457>

- Server-side image processing, storage and OCR.
- Scalable, distributed infrastructure with remote monitoring, automated CI/CD pipeline and containerized deployment.
- Multiple thread mitigation techniques, static code analysis, unit testing and fuzzing to enhance security and software reliability.

The main objective was to develop a fully functional infrastructure. Additionally, significant focus was placed on security through various techniques to identify and mitigate vulnerabilities.

2 System Architecture

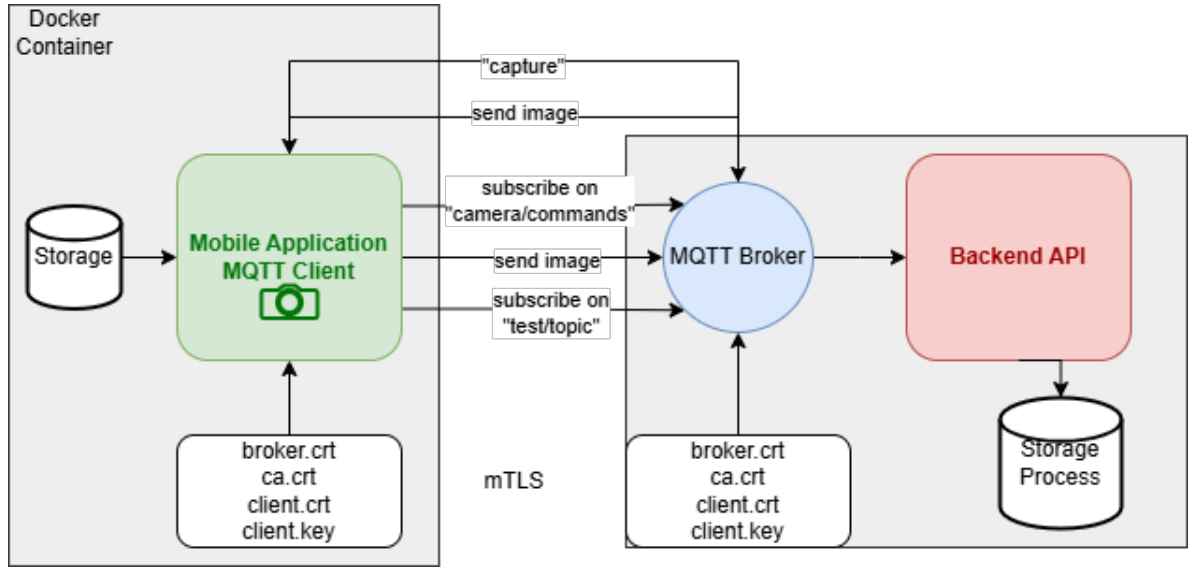


Figure 1: MQTTImage Project Architecture

Figure 1 outlines the main elements of the system architecture. On one side, the mobile application transmits images via MQTT over mutual TLS (mTLS), using two distinct topics to support different modes of operation: classic scheduled capture, live streaming and on-demand requests, as well as local image storage for offline scenarios.

On the other side, the images are routed through the MQTT broker to the backend API, where they are securely processed and stored. The entire system is containerized, with mirrored certificate-based authentication ensuring end-to-end secure communication and infrastructure isolation.

3 Implemented Features

3.1 Secure Communication via mTLS

To ensure secure, authenticated communication between the mobile application and the backend system, mutual TLS (mTLS) is used over MQTT. The proper generation, management and integration of client and server certificates into the Android application was one of the most difficult technical challenges of the project. It required careful handling

of certificate formats, secure storage within the app and smooth integration into the SSL/TLS handshake to ensure strong and reliable end-to-end security.

The certificates were generated using resources such as Mosquitto MQTT mTLS guide and MQTT async client mTLS example. The challenges encountered were diverse, ranging from Certificate Authority (CA) errors and issues with Subject Alternative Name (SAN) configurations to port setup complexities. Below is a brief series of useful commands for the final steps in generating certificates for the CA, broker and client through OpenSSL³:

Listing 1: Example commands for certificate generation

```
1 # Generate CA key and self-signed certificate
2 openssl req -new -x509 -days 365 -keyout ca.key -out ca.crt
3
4 # Generate broker private key and certificate signing request (
   CSR)
5 openssl genrsa -out broker.key 2048
6 openssl req -new -key broker.key -out broker.csr
7
8 # Sign broker CSR with CA to create broker certificate
9 openssl x509 -req -in broker.csr -CA ca.crt -CAkey ca.key -
   CAcreateserial -out broker.crt -days 200
10
11 # Generate client private key and certificate signing request (
   CSR)
12 openssl genrsa -out client.key 2048
13 openssl req -new -key client.key -out client.csr
14
15 # Sign client CSR with CA to create client certificate
16 openssl x509 -req -in client.csr -CA ca.crt -CAkey ca.key -
   CAcreateserial -out client.crt -days 200
```

For the MQTT broker, we used Mosquitto⁴ with the following configuration file snippet (Listing 2) to enable TLS with mutual authentication. The Mosquitto broker is started using the command: `mosquitto -c mosquitto.conf -d`, which runs the broker in daemon mode with the specified configuration file.

For testing purposes, clients can connect using the `mosquitto_sub` command, for example:

```
mosquitto_sub -h 192.168.1.110 -p 8883 -t "test/topic" --cafile certs/ca.crt
--cert certs/broker.crt --key certs/broker.key -d --tls-version tlsv1.2
```

However, in this project, our focus is primarily on the Android client, which connects securely to the broker using the generated certificates and mutual TLS.

Listing 2: MQTT Broker mosquitto.conf

```
1 # Listen on port 1883 (non-secure) for local connections
2 listener 1883
3 allow_anonymous true
4
5 # Listen on port 8883 for secure (TLS) connections
```

³<https://www.openssl.org>

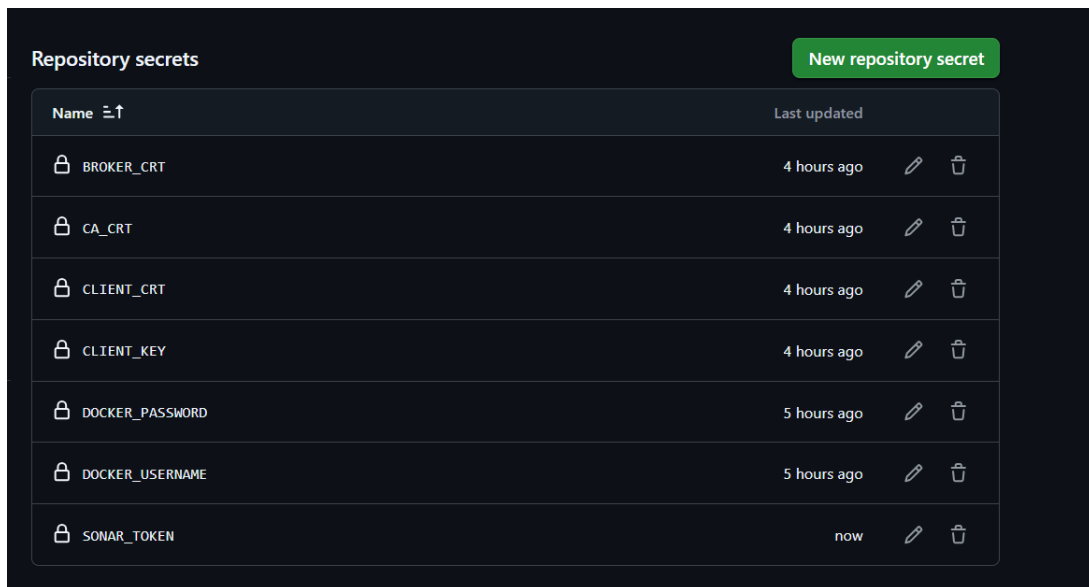
⁴<https://mosquitto.org>

```

6 listener 8883 0.0.0.0
7 allow_anonymous true
8
9 # TLS/SSL certificate files
10 cafile /mosquitto/config/certs/ca.crt
11 certfile /mosquitto/config/certs/broker.crt
12 keyfile /mosquitto/config/certs/broker.key
13
14 # Enforce the TLS version (stronger security with TLSv1.2)
15 tls_version tlsv1.2
16
17 log_type all

```

Throughout the project, we used GitHub CI/CD⁵ pipelines to automate builds and deployments. Importantly, the certificates were never directly committed to the repository. Instead, they were handled securely as raw resources only during the APK generation process. Additionally, the certificates required for Mosquitto’s configuration were stored and managed securely within GitHub Secrets, ensuring they remain confidential throughout the CI/CD pipeline.



Repository secrets		New repository secret
Name	Last updated	
BROKER_CERT	4 hours ago	
CA_CERT	4 hours ago	
CLIENT_CERT	4 hours ago	
CLIENT_KEY	4 hours ago	
DOCKER_PASSWORD	5 hours ago	
DOCKER_USERNAME	5 hours ago	
SONAR_TOKEN	now	

Figure 2: GitHub Secrets for Confidential Certificates in CI/CD

3.2 Android Implementation

3.2.1 Compatibility and Permissions

The Android application serves as the primary client for capturing and securely transmitting images to the backend via MQTT over mutual TLS (mTLS). It was developed from scratch in Java (JDK17⁶) as the primary client for capturing and securely transmitting images to the backend via MQTT over mutual TLS (mTLS). It is compatible with de-

⁵<https://github.blog/enterprise-software/ci-cd/build-ci-cd-pipeline-github-actions-four-steps/>

⁶<https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html>

vices running Android SDK versions with a minimum SDK version of 26 (Android 8 ⁷), targeting SDK version 33 (Android 13 ⁸), build using Gradle⁹.

The application requires several permissions to operate correctly and securely. These include `INTERNET` permission for network communication, `CAMERA` permission to capture images, `ACCESS_NETWORK_STATE` to monitor network connectivity, `WAKE_LOCK` to keep the CPU awake during critical operations and `RECEIVE_BOOT_COMPLETED` to enable automatic initialization after device reboot. Runtime permission requests are implemented for sensitive operations, such as accessing the camera, ensuring compliance with Android security best practices and enhancing user privacy.

3.2.2 Network Connectivity Management and MQTT Security

The `NetworkMonitor` class manages network connectivity changes by leveraging Android's `ConnectivityManager` and `NetworkRequest` APIs. It monitors Wi-Fi network availability and automatically establishes or terminates MQTT connections accordingly. Upon network availability, it triggers the MQTT client to connect to the broker and sets up a message listener to handle commands such as capturing images on demand. Conversely, it disconnects the MQTT client when the network is lost, and upon network restoration, it automatically resends any images that were saved locally during the offline period. This design ensures robust and dynamic management of the MQTT communication channel in response to fluctuating network conditions, enhancing application reliability and responsiveness.

The Android application leverages the Eclipse Paho MQTT¹⁰ client library to establish and maintain encrypted connections with the broker, securely loading certificates and private keys from the application's raw resources. The implementation uses BouncyCastle to parse and decrypt PEM-formatted keys, enabling the creation of a custom SSL socket factory that manages client authentication and server trust verification. The key function responsible for loading the CA certificate, client certificate, private key and for creating a properly configured `SSLConnectionFactory` for mutual TLS, is detailed in Appendix A.

Once connected, the Android MQTT client subscribes to two topics, `"camera/commands"` and `"test/topic"`, as illustrated in Figure 1. This allows the client to receive commands from the backend, such as triggering image captures and to publish or receive image data and status updates accordingly.

3.2.3 Functionalities

The main activity screen features two primary buttons and a mode selection spinner (Figure 3). The **Capture Image** button initiates the camera to take a photo, while the **Delete Local Images** button allows the user to clear all images stored locally when offline or not connected to MQTT broker. Additionally, a spinner control enables the user to select the application's operational mode, offering options such as:

- **NONE:** Each captured image is sent to the MQTT topic `test/topic` immediately after capture.

⁷<https://apilevels.com>

⁸<https://apilevels.com>

⁹<https://docs.gradle.org/current/userguide/compatibility.html>

¹⁰<https://github.com/eclipse-paho/paho.mqtt.java>

- **ON_DEMAND:** Images are captured and sent only when explicitly requested by the user or via MQTT "capture" command on camera/commands topic.
- **PERIODIC:** The app captures and sends all stored images automatically with a timer of 30 seconds.
- **LIVE:** Images are captured and sent continuously every second to provide a live feed.

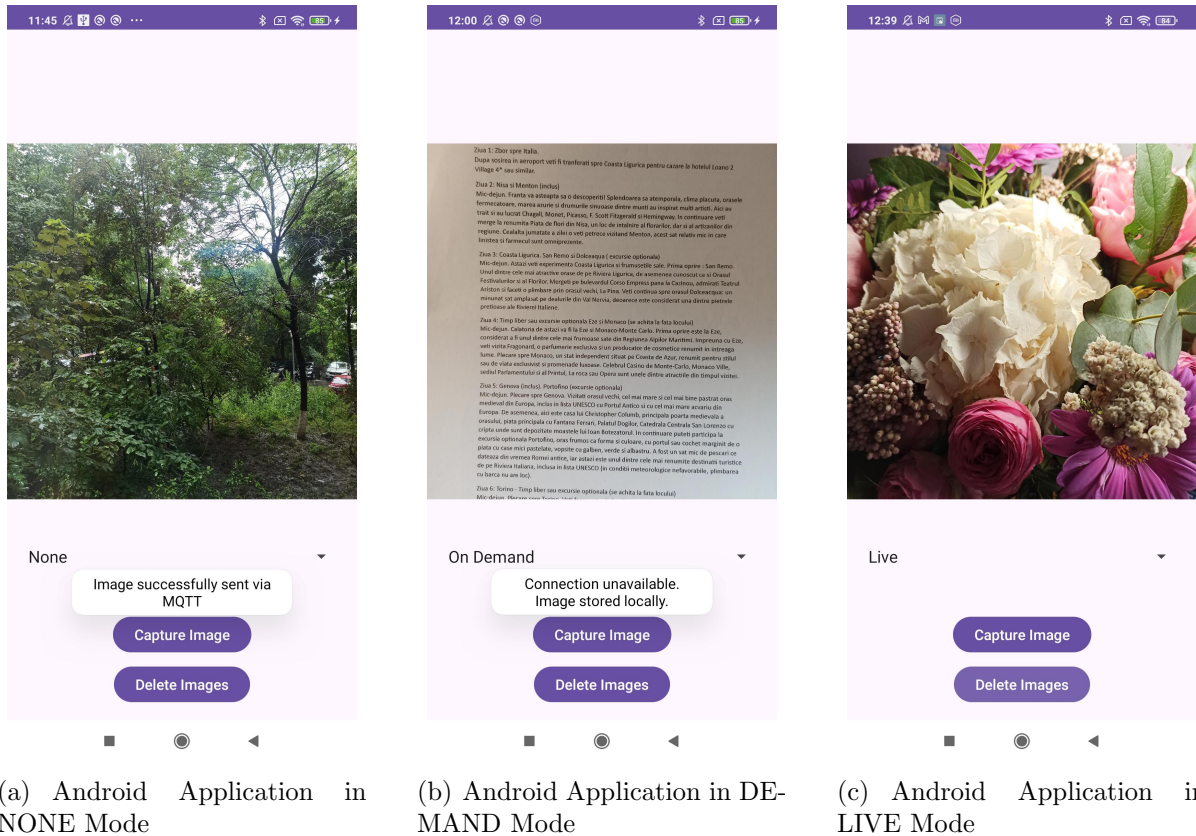


Figure 3: MQTTImage Application in Various Modes

Figure 4 illustrates the reception and processing of images from the MQTT broker's perspective, as established in the architecture shown in Figure 1. It definitely can be observed that in **PERIODIC Mode**, images are sent every 30 seconds, while in **LIVE Mode**, they are sent every 1 second. When the connection is lost, images are stored locally. In **ON_DEMAND Mode**, if another client sends a "capture" command to the "camera/commands" topic, the locally stored images are transmitted immediately.

```
c:\Program Files\mosquitto>mosquitto -c certs_git2.conf -d
1748249002: Warning: Can't start in daemon mode in Windows.
1748249002: mosquitto version 2.0.20 starting
1748249002: Config loaded from certs_git2.conf.
1748249002: Opening ipv6 listen socket on port 1883.
1748249002: Opening ipv4 listen socket on port 1883.
1748249002: Opening ipv4 listen socket on port 8883.
1748249002: mosquitto version 2.0.20 running
1748249002: New connection from 192.168.1.103:59446 on port 8883.
1748249002: New client connected from 192.168.1.103:59446 as Android_client (p2, c1, k60).
1748249002: No will message specified.
1748249002: Sending CONNACK to Android_client (0, 0)
1748249062: Received PINGREQ from Android_client
1748249062: Sending PINGRESP to Android_client
1748249109: Received PUBLISH from Android_client (d0, q1, r0, m2, 'test/topic', ... (10705215 bytes))
1748249109: Sending PUBACK to Android_client (m2, rc0)
```

(a) MQTT Broker in NONE Mode

```
1748252430: Sending PUBACK to Android_client (m35, rc0)
1748252430: Received PUBLISH from Android_client (d0, q1, r0, m36, 'test/topic', ... (927921 bytes))
1748252430: Sending PUBACK to Android_client (m36, rc0)
1748252431: Received PUBLISH from Android_client (d0, q1, r0, m37, 'test/topic', ... (927921 bytes))
1748252431: Sending PUBACK to Android_client (m37, rc0)
1748252431: Received PUBLISH from Android_client (d0, q1, r0, m38, 'test/topic', ... (927921 bytes))
1748252431: Sending PUBACK to Android_client (m38, rc0)
1748252431: Received PUBLISH from Android_client (d0, q1, r0, m39, 'test/topic', ... (927921 bytes))
1748252431: Sending PUBACK to Android_client (m39, rc0)
1748252432: Received PUBLISH from Android_client (d0, q1, r0, m40, 'test/topic', ... (927921 bytes))
1748252432: Sending PUBACK to Android_client (m40, rc0)
1748252432: Received PUBLISH from Android_client (d0, q1, r0, m41, 'test/topic', ... (927921 bytes))
1748252432: Sending PUBACK to Android_client (m41, rc0)
1748252433: Received PUBLISH from Android_client (d0, q1, r0, m42, 'test/topic', ... (927921 bytes))
1748252433: Sending PUBACK to Android_client (m42, rc0)
1748252433: Received PUBLISH from Android_client (d0, q1, r0, m43, 'test/topic', ... (927921 bytes))
1748252433: Sending PUBACK to Android_client (m43, rc0)
1748252434: Received PUBLISH from Android_client (d0, q1, r0, m44, 'test/topic', ... (927921 bytes))
1748252434: Sending PUBACK to Android_client (m44, rc0)
1748252434: Received PUBLISH from Android_client (d0, q1, r0, m45, 'test/topic', ... (927921 bytes))
1748252434: Sending PUBACK to Android_client (m45, rc0)
1748252435: Received PUBLISH from Android_client (d0, q1, r0, m46, 'test/topic', ... (927921 bytes))
1748252435: Sending PUBACK to Android_client (m46, rc0)
1748252435: Received PUBLISH from Android_client (d0, q1, r0, m47, 'test/topic', ... (927921 bytes))
1748252435: Sending PUBACK to Android_client (m47, rc0)
1748252436: Received PUBLISH from Android_client (d0, q1, r0, m48, 'test/topic', ... (927921 bytes))
1748252436: Sending PUBACK to Android_client (m48, rc0)
1748252436: Received PUBLISH from Android_client (d0, q1, r0, m49, 'test/topic', ... (927921 bytes))
1748252436: Sending PUBACK to Android_client (m49, rc0)
1748252437: Received PUBLISH from Android_client (d0, q1, r0, m50, 'test/topic', ... (927921 bytes))
1748252437: Sending PUBACK to Android_client (m50, rc0)
1748252437: Received PUBLISH from Android_client (d0, q1, r0, m51, 'test/topic', ... (927921 bytes))
1748252437: Sending PUBACK to Android_client (m51, rc0)
1748252437: Received PUBLISH from Android_client (d0, q1, r0, m52, 'test/topic', ... (927921 bytes))
1748252437: Sending PUBACK to Android_client (m52, rc0)
1748252438: Received PUBLISH from Android_client (d0, q1, r0, m53, 'test/topic', ... (927921 bytes))
1748252438: Sending PUBACK to Android_client (m53, rc0)
1748252438: Received PUBLISH from Android_client (d0, q1, r0, m54, 'test/topic', ... (927921 bytes))
1748252438: Sending PUBACK to Android_client (m54, rc0)
```

(b) MQTT Broker in LIVE Mode

```
Qc periods
2025-05-26 12:34:08.439 30254-30290 MQTT com.example.ss_final.java D Connected to broker: ssl://192.168.1.110:8883
2025-05-26 12:34:08.439 30254-30290 MQTT com.example.ss_final.java D Subscribing to topic: camera/commands
2025-05-26 12:34:08.451 30254-30290 MQTT com.example.ss_final.java D Subscribed to topic: camera/commands
2025-05-26 12:34:24.647 30254-30254 MQTT com.example.ss_final.java W ERROR [MQTT]: /data/user/0/com.example.ss_final.java/files/offline_images
2025-05-26 12:34:24.662 30254-30254 MQTT com.example.ss_final.java D Binary message published to topic: test/topic
2025-05-26 12:34:24.662 30254-30254 MQTT com.example.ss_final.java D Image saved locally: img_1748252004057.jpg
2025-05-26 12:34:26.072 30254-30254 System.err com.example.ss_final.java W ERROR [MQTT]: /data/user/0/com.example.ss_final.java/files/offline_images
2025-05-26 12:34:26.072 30254-30254 MQTT com.example.ss_final.java D Binary message published to topic: test/topic
2025-05-26 12:34:26.072 30254-30254 MQTT com.example.ss_final.java D Image saved locally: img_1748252009008.jpg
2025-05-26 12:34:29.077 30254-30254 System.err com.example.ss_final.java W ERROR [MQTT]: /data/user/0/com.example.ss_final.java/files/offline_images
2025-05-26 12:34:29.077 30254-30254 MQTT com.example.ss_final.java D Binary message published to topic: test/topic
2025-05-26 12:34:29.077 30254-30254 MQTT com.example.ss_final.java D Image saved locally: img_1748252100406.jpg
2025-05-26 12:35:00.438 30254-30254 MQTT com.example.ss_final.java D Binary message published to topic: test/topic
2025-05-26 12:35:00.438 30254-30254 MQTT com.example.ss_final.java D Image saved locally: img_1748252100406.jpg
2025-05-26 12:35:04.255 30254-30254 MQTT com.example.ss_final.java D Binary message published to topic: test/topic
2025-05-26 12:35:04.255 30254-30254 MQTT com.example.ss_final.java D Image saved locally: img_1748252104261.jpg
2025-05-26 12:35:26.073 30254-30254 System.err com.example.ss_final.java W ERROR [MQTT]: /data/user/0/com.example.ss_final.java/files/offline_images
2025-05-26 12:35:26.073 30254-30254 MQTT com.example.ss_final.java D Binary message published to topic: test/topic
2025-05-26 12:35:31.992 30254-30254 MQTT com.example.ss_final.java D Image saved locally: img_1748252132020.jpg
2025-05-26 12:35:32.055 30254-30254 MQTT com.example.ss_final.java D Binary message published to topic: test/topic
2025-05-26 12:35:37.530 30254-30254 MQTT com.example.ss_final.java D Image saved locally: img_1748252137554.jpg
2025-05-26 12:35:37.580 30254-30254 MQTT com.example.ss_final.java
```

(c) MQTT Broker in PERIODIC Mode

```
1748250000: No will message specified.
1748250000: Sending CONNACK to auto-D41A4E8F-A3B3-A76F-C8AB-4CC954728831 (0, 0)
1748250000: Received PUBLISH from auto-D41A4E8F-A3B3-A76F-C8AB-4CC954728831 (d0, q0, r0, m0, 'camera/commands', ... (7 bytes))
1748250000: Sending PUBLISH to Android_client (d0, q0, r0, m0, 'camera/commands', ... (7 bytes))
1748250000: Received DISCONNECT from auto-D41A4E8F-A3B3-A76F-C8AB-4CC954728831
1748250000: client auto-D41A4E8F-A3B3-A76F-C8AB-4CC954728831 disconnected.
1748250002: Received PUBLISH from Android_client (d0, q1, r0, m3, 'test/topic', ... (10705215 bytes))
1748250002: Sending PUBACK to Android_client (m3, rc0)
1748250007: Received PUBLISH from Android_client (d0, q1, r0, m4, 'camera/commands', ... (10705215 bytes))
1748250007: Sending PUBLISH to Android_client (d0, q0, r0, m0, 'camera/commands', ... (10705215 bytes))
1748250007: Sending PUBACK to Android_client (m4, rc0)
c:\Program Files\mosquitto>mosquitto_pub -h 192.168.1.110 -p 8883 -t "camera/command s" -m "capture" --cafile certs_git2/ca.crt --cert certs_git2/broker.crt --key certs_git2/broker.key --tls-version tlsv1.2
c:\Program Files\mosquitto>mosquitto_pub -h 192.168.1.110 -p 8883 -t "camera/command s" -m "capture" --cafile certs_git2/ca.crt --cert certs_git2/broker.crt --key certs_git2/broker.key --tls-version tlsv1.2
c:\Program Files\mosquitto>mosquitto_pub -h 192.168.1.110 -p 8883 -t "camera/command s" -m "capture" --cafile certs_git2/ca.crt --cert certs_git2/broker.crt --key certs_git2/broker.key --tls-version tlsv1.2
c:\Program Files\mosquitto>mosquitto_pub -h 192.168.1.110 -p 8883 -t "camera/command s" -m "capture" --cafile certs_git2/ca.crt --cert certs_git2/broker.crt --key certs_git2/broker.key --tls-version tlsv1.2
```

(d) MQTT Broker in DEMAND Mode

Figure 4: MQTT Broker in Various Modes

3.2.4 Android Libraries

The project uses AndroidX libraries for UI components and layout management, including Material Design¹¹ and ConstraintLayout¹². For camera functionality, it relies on CameraX libraries for easy and lifecycle-aware camera integration. MQTT communication is handled with Eclipse Paho libraries, enabling secure message exchange. Cryptographic operations, especially for SSL/TLS, use BouncyCastle. Testing is supported by JUnit¹³ and Mockito¹⁴ for unit tests, along with AndroidX testing libraries and Robolectric¹⁵ for running tests on the JVM. Additionally, Jacoco¹⁶ is used for code coverage, and SonarQube¹⁷ ensures code quality through static analysis.

3.3 Backend System

The backend system is implemented using Python and Flask, functioning both as an HTTP API server and an MQTT client over mutual TLS. It securely connects to the Mosquitto MQTT broker using client certificates and listens for image messages published to the topic `"test/topic"`.

Upon receiving a binary image payload, the backend saves the image to a local folder and performs Optical Character Recognition (OCR) using the Tesseract OCR engine. The extracted text is logged for further use or debugging. This integration provides automated server-side processing of image content transmitted from the Android client.

The backend is implemented in a single file, `app.py`, which encapsulates Flask route handling, MQTT configuration with mTLS, and OCR extraction. It uses the Paho MQTT client for subscribing to messages securely, the Pillow library for image handling, and pytesseract for text extraction.

The required Python dependencies are specified in a separate `requirements.txt` file and include `flask`, `paho-mqtt`, `pillow`, and `pytesseract`. Tesseract-OCR is also installed system-wide.

The figure below shows the Flask backend running and successfully receiving an image from the MQTT pipeline:

¹¹<https://material.io/design>

¹²<https://developer.android.com/develop/ui/views/layout/constraint-layout>

¹³<https://junit.org/junit5/>

¹⁴<https://site.mockito.org>

¹⁵<https://robolectric.org>

¹⁶<https://www.jacoco.org/jacoco/trunk/index.html>

¹⁷<https://www.sonarsource.com/products/sonarqube/>


```
venv) [x]-[skar@parrot]-[~/Documents/Facultate/SS/project/SSProject/flask_backend]
$python app.py
home/skar/Documents/Facultate/SS/project/SSProject/flask_backend/app.py:53: DeprecationWarning: Callback API version 1 is deprecated, update to latest version
mqtt_client = Client()
INFO:mqtt_flask_backend:[MQTT] Connected to localhost:1883
INFO:mqtt_flask_backend:[MQTT] Subscribed to topic: test/topic
* Serving Flask app 'app'
* Debug mode: off
INFO:werkzeug:WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://192.168.1.134:5000
INFO:werkzeug:Press CTRL+C to quit
INFO:mqtt_flask_backend:[MQTT] Message received on topic: test/topic
INFO:mqtt_flask_backend:[MQTT] Image saved as received_images/image_20250526_1011.jpg
```

Figure 5: Flask Backend Receiving and Processing Image via MQTT over mTLS

The relevant steps performed by the backend upon receiving an MQTT message are:

1. Establish secure MQTT connection using TLS and client certificates.
2. Subscribe to the topic `test/topic`.
3. Upon receiving a binary payload, save it as a timestamped JPEG file.
4. Open the saved image and extract text using Tesseract OCR.
5. Log the extracted text and save the image for future access.

3.4 CI/CD Pipeline and Docker

For this project, we have established a GitHub repository to manage version control efficiently. The repository follows a structured branching strategy to support collaborative development and maintain code quality. This setup enables seamless integration with CI/CD pipelines, ensuring automated building, testing, and deployment of our Dockerized application.

We have created Dockerfiles for both the Android application and the Mosquitto MQTT broker to containerize the components effectively. Additionally, a `docker-compose.yml` file is used to orchestrate and run these services locally, simplifying development and testing by managing multiple containers with a single command. The setup also supports automated build and push processes to container registries like Docker Hub (<https://hub.docker.com/repositories/denisapopescu1905>) or GitHub Container Registry for streamlined deployment.

Listing 3: Project folder structure with Docker and CI/CD

```

SS_Final_Java/
|
+-- android_app/          # Code and Dockerfile for Android App
|   |
|   +--app/
|   +--Dockerfile
|   +--docker-compose.yml
|-- mqtt-pipeline/mqtt    # Code and Dockerfile for Mosquitto
|   +--certs/
|   +--Dockerfile
|   +--mosquitto.conf
|   +--docker-compose.yml
|
+-- .github/workflows
|   +--android.yml        # CI/CD Build for Android and Mosquitto
|   +--build.yml          # Sonar

```

This GitHub Actions workflow automates the CI/CD process for the project elements using Docker. It triggers on pushes or pull requests to the main branch. On short, the steps include:

- Automates CI/CD pipeline for Android app and MQTT broker with Docker.
- Triggers on pushes or pull requests to the main branch.
- Checks out code and sets up Docker Buildx.
- Logs into Docker Hub securely using secrets.
- Configures SSL/TLS certificates for MQTT broker.
- Builds and pushes Docker images for Android app and MQTT broker.
- Sets up Java environment and makes Gradle wrapper executable.
- Runs static code analysis with SonarQube to ensure code quality.
- Uploads the generated APK artifact for later use or download.

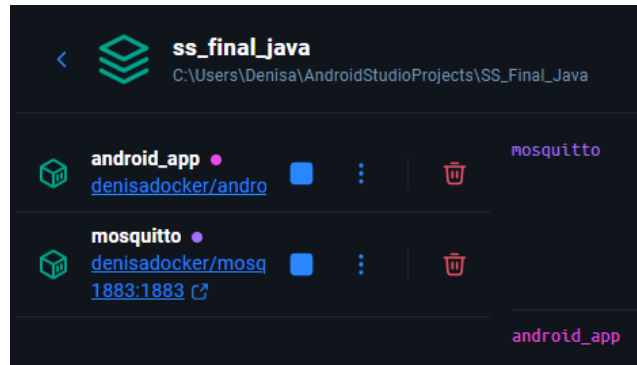
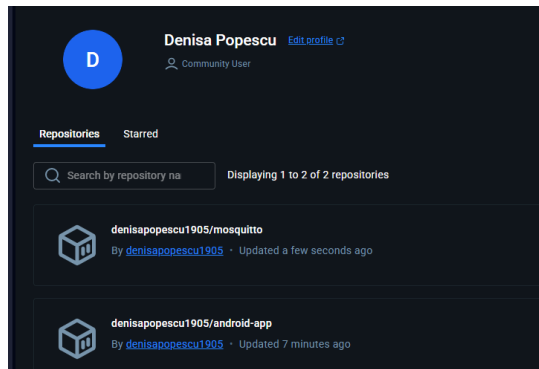


Figure 6: Docker Repositories

```
C:\Users\Denisa\AndroidStudioProjects\SS_Final_Java>docker-compose up --build
time="2025-05-25T13:58:56+03:00" level=warning msg="C:\Users\Denisa\AndroidStudioProjects\SS_Final_Java\docker-compose.yml: the attribute 'version' is obsolete, it will be ignored, please remove it to avoid potential confusion"
Compose can now delegate builds to bake for better performance.
To do so, set COMPOSE_BAKE=true.
2025/05/25 13:58:56 http: server: error reading preface from client //.: /pipe/dockerDesktopLinuxEngine: file has already been closed
[+] Building 1.1s (28/28) FINISHED
    => [mosquitto internal] load build definition from Dockerfile
    => -- transferring dockerfile: 157B
    => [android_app internal] load build definition from Dockerfile
    => -- transferring dockerfile: 1.39kB
    => [android_app internal] load metadata for docker.io/library/openjdk:17-slim
    => [mosquitto internal] load metadata for docker.io/library/eclipse-mosquitto:2.0
    => [android_app auth] library/openjdk:pull token for registry-1.docker.io
    => [mosquitto auth] library/eclipse-mosquitto:pull token for registry-1.docker.io
    => [android_app internal] load .dockerignore
    => -- transferring context: 2B
    => [mosquitto internal] load .dockerignore
    => -- transferring context: 2B
    => [android_app 1/11] FROM docker.io/library/openjdk:17-slim@sha256:aaa3b3cb27e3e520b8f116863d0580c438ed5ecfa0
    => [android_app internal] load build context
    => -- transferring context: 6.03kB
    => [mosquitto 1/1] FROM docker.io/library/eclipse-mosquitto:2.0@sha256:94f5a3d7deafa59fa3448d227ddad558f59d293c6
    => [mosquitto internal] load build context
    => -- transferring context: 201B
    => CACHED [mosquitto 2/3] COPY mosquitto.conf /mosquitto/config/mosquitto.conf
    => CACHED [mosquitto 3/3] COPY certs/ /mosquitto/config/certs/
    => [mosquitto] exporting to image
    => -- exporting layers
    => -- writing image sha256:a404db7fd05923f3c0b03ef0967f39e74c6a61ef9cb3d4e9dfa58528ce8d0
    => naming to docker.io/denisadocker/mosquitto:latest
    => CACHED [android_app 2/11] RUN apt-get update && apt-get install -y wget unzip
    => CACHED [android_app 3/11] RUN mkdir -p /sdk/cmdline-tools
    => CACHED [android_app 4/11] RUN wget https://dl.google.com/android/repository/commandlinetools-linux-9477386.l
    => CACHED [android_app 5/11] RUN mkdir -p /sdk/cmdline-tools/latest && mv /sdk/cmdline-tools/cmdline-tools/
    => CACHED [android_app 6/11] RUN yes | sdkmanager --licenses
    => CACHED [android_app 7/11] RUN sdkmanager "platform-tools" "platforms;android-33" "build-tools;33.0.0"
    => CACHED [android_app 8/11] WORKDIR /app
    => CACHED [android_app 9/11] COPY .
    => CACHED [android_app 10/11] RUN chmod +x ./gradlew
    => CACHED [android_app 11/11] RUN ./gradlew assembleDebug
    => [android_app] exporting to image
    => -- exporting layers
    => -- writing image sha256:00c9223cecf3a07dca47101b4dec1430000e4175b93342156c0d8e238239cda
    => naming to docker.io/denisadocker/android-app:latest
    => [mosquitto] resolving provenance for metadata file
    => [android_app] resolving provenance for metadata file
[+] Running 5/5
  ✓ android_app      Built
  ✓ mosquitto        Built
  ✓ network ss_final_java_default Created
  ✓ container ss_final_java-mosquitto-1 Created
  ✓ container ss_final_java-android-app-1 Created
Attaching to android_app-1, mosquitto-1
mosquitto-1 | 1748170738: mosquitto version 2.0.21 starting
mosquitto-1 | 1748170738: Config loaded from /mosquitto/config/mosquitto.conf.
mosquitto-1 | 1748170738: Opening ipv4 listen socket on port 1883.
mosquitto-1 | 1748170738: Opening ipv6 listen socket on port 1883.
mosquitto-1 | 1748170738: Opening ipv4 listen socket on port 8883.
mosquitto-1 | 1748170738: mosquitto version 2.0.21 running
android_app-1 | May 25, 2025 10:58:58 AM java.util.prefs.FileSystemPreferences$1 run
android_app-1 | INFO: Created user preferences directory.
android_app-1 | Welcome to JShell -- Version 17.0.2
android_app-1 | For an introduction type: /help Intro
```

Figure 7: Docker Build

Figure 6 shows the Docker images successfully pushed to the Docker Desktop repository. Figure 7 presents a build of the Android application, confirming that the image compiles and packages correctly. Figure 8 demonstrates the CI/CD integration in action, where the pipeline is triggered automatically on every commit. All credentials and certificates, including Docker Hub credentials and SonarQube tokens, are securely managed through

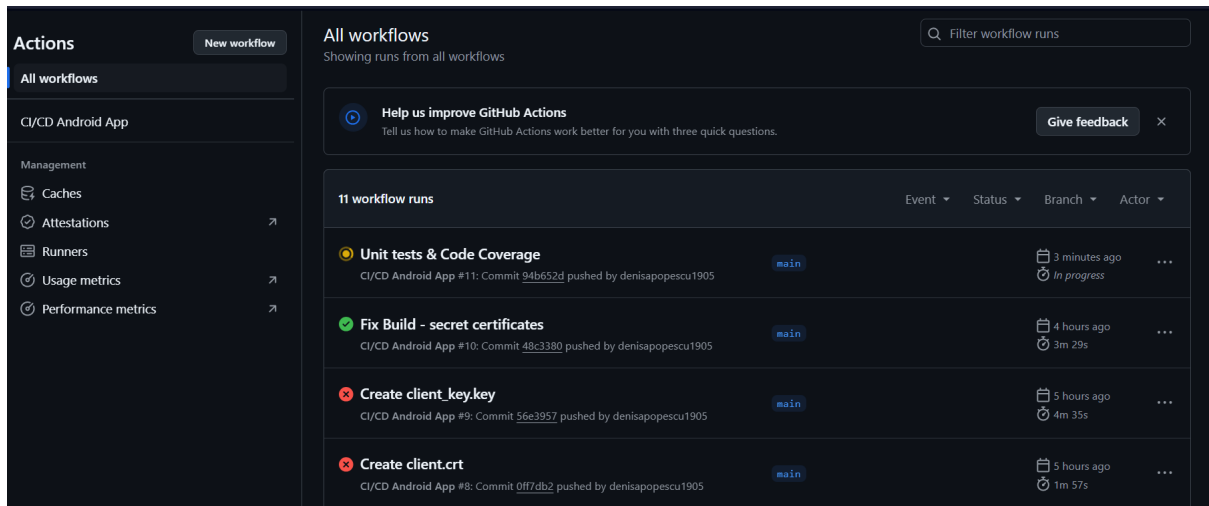


Figure 8: CI/CD Github Actions Workflow

GitHub Secrets, enabling future integration with static analysis and code quality tools such as SonarQube.

4 Security And Compliance

In this project, special attention was given to security and compliance aspects. We aimed to identify potential risks early, mitigate vulnerabilities and ensure a robust and trustworthy system. This included analyzing possible threats, reviewing the quality and safety of the code and verifying that all components used were secure and up to date. Additionally, we maintained good practices by tracking and resolving issues—both in our own project and in peer reviews—demonstrating a proactive approach to building secure and reliable software.

4.1 Unit Testing and Code Coverage

As part of the development process, we focused on ensuring software quality through unit testing and code coverage analysis. Key components of the application were tested in isolation to validate their correctness and robustness. By measuring how much of the code was exercised by the tests, we identified areas that required better coverage, helping to reduce the risk of undetected bugs. These tests were integrated into the CI/CD pipeline, allowing automated execution at each commit or pull request. This approach ensures consistent quality checks and supports early detection of issues, contributing to the reliability and maintainability of the project.

On one hand, unit tests were implemented using JUnit, Mockito and RobolectricTestRunner (Figure 9). The main goal was to validate application behavior, simulate external dependencies and detect logic flaws early in the development lifecycle. Robolectric enables running Android framework-dependent tests on the JVM without an emulator, providing faster and more isolated tests for UI and lifecycle components. The unit tests have a key testing strategy such as:

- Activity lifecycle methods such as `startPeriodicMode`, `stopLiveMode`, and `stopPeri-`

odicMode were validated for correct invocation.

- MQTT client behavior was thoroughly tested, including publish/subscribe mechanisms, binary data handling, reconnect logic, and error scenarios.
- Offline data storage was verified to ensure unsent images are locally stored and later resent when connectivity is restored.
- Exception handling was tested to confirm graceful fallback during network errors or certificate failures.
- Message listener integration was covered, checking that incoming MQTT messages are correctly captured and processed.
- Network monitoring ensured that loss of connectivity correctly triggers disconnect routines via the NetworkMonitor class.
- Code coverage was improved by simulating both success and failure paths, with targeted use of mocking and argument capturing.

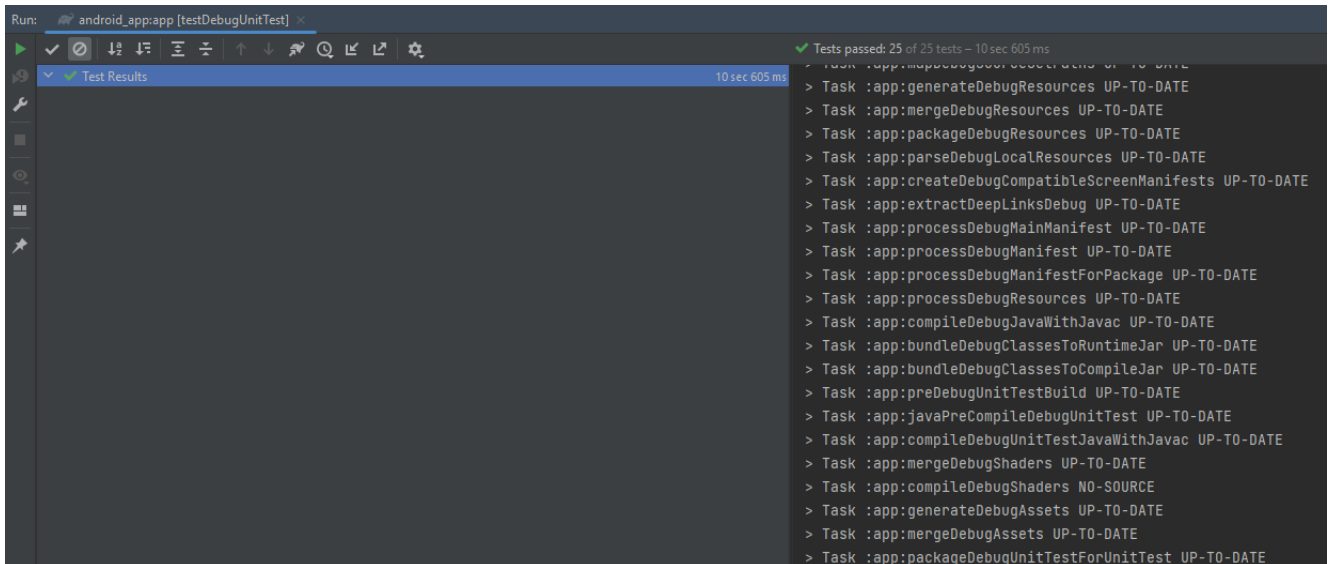


Figure 9: Unit Testing

Before running the test suite, we set a code coverage threshold of 40%. JaCoCo was used to generate the code coverage report, demonstrating that the target was met. In the future, we aim to increase the coverage threshold to 60% for improved code quality.¹⁰ As the report shows, the areas that could be improved are those related to NetworkMonitor and timers.

4.2 Static Code Analysis

For this Java project, SonarCloud was used to perform static code analysis. It is integrated with the GitHub CI/CD pipeline to automatically scan the code at each build. This setup helps detect vulnerabilities and maintain code quality continuously throughout development.

app > com.example.ss_final_java [Source Files](#) [Sessions](#)

com.example.ss_final_java

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Clas
MainActivity	<div><div></div></div>	6%	<div><div></div></div>	8%	28	32	100	109	11	15	0	
MqttHandler	<div><div></div></div>	64%	<div><div></div></div>	59%	16	35	53	155	2	14	0	
NetworkMonitor.new ConnectivityManager.NetworkCallback().{...}	<div><div></div></div>	0%	<div><div></div></div>	0%	7	7	16	16	4	4	1	
MainActivity.new AdapterView.OnItemClickListener().{...}	<div><div></div></div>	0%	<div><div></div></div>	0%	7	7	20	20	3	3	1	
NetworkMonitor	<div><div></div></div>	0%	<div><div></div></div>	n/a	3	3	14	14	3	3	1	
MainActivity.new TimerTask().{...}	<div><div></div></div>	0%	<div><div></div></div>	n/a	3	3	6	6	3	3	1	
MainActivity.new TimerTask().{...}	<div><div></div></div>	59%	<div><div></div></div>	n/a	1	3	3	6	1	3	0	
Logger	<div><div></div></div>	85%	<div><div></div></div>	50%	2	6	2	9	1	5	0	
ToastWrapper	<div><div></div></div>	84%	<div><div></div></div>	50%	1	4	1	12	0	3	0	
MainActivity.Mode	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1	0	
Total		871 of 1,482	41%	61 of 91	32%	68	101	212	344	28	54	4

Created with JaCoCo 0.8.8.202204050719

Figure 10: JaCoco Code Coverage

4.2.1 SonarQube Reported Issues

SonarCloud automatically identifies code quality and security issues during the CI/CD builds. It highlights bugs, code smells, and potential vulnerabilities based on secure CERT Java guidelines¹⁸, helping us prioritize fixes and improve the overall reliability and maintainability of the project.

The first SonarCloud analysis (Figure 11) presents the distribution of issues by severity, offering a clear overview of the most critical code quality and security concerns. It also defines an initial attack surface consisting of 18 identified vulnerabilities (Figure 12), helping guide future mitigation efforts.

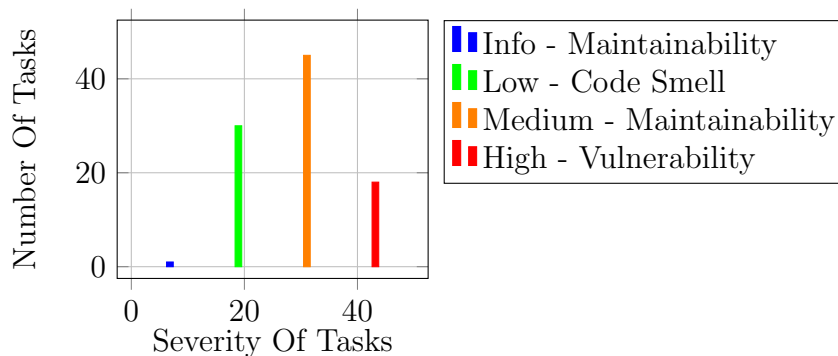


Figure 11: SonarCloud Static Analysis Summary

Additionally, 21 issues were also reported by the Lint Report in Figure 13.

4.3 Threat Modeling And Mitigations

Based on the discovered attack surface, we have analyzed the vulnerabilities and possible mitigations, all presented in Table 1. From a security perspective, the most critical issues are those related to disabled hostname verification and the lack of code obfuscation, especially relevant when releasing the application to production. These vulnerabilities pose significant risks such as Man-in-the-Middle attacks and reverse engineering threats.

¹⁸<https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java>

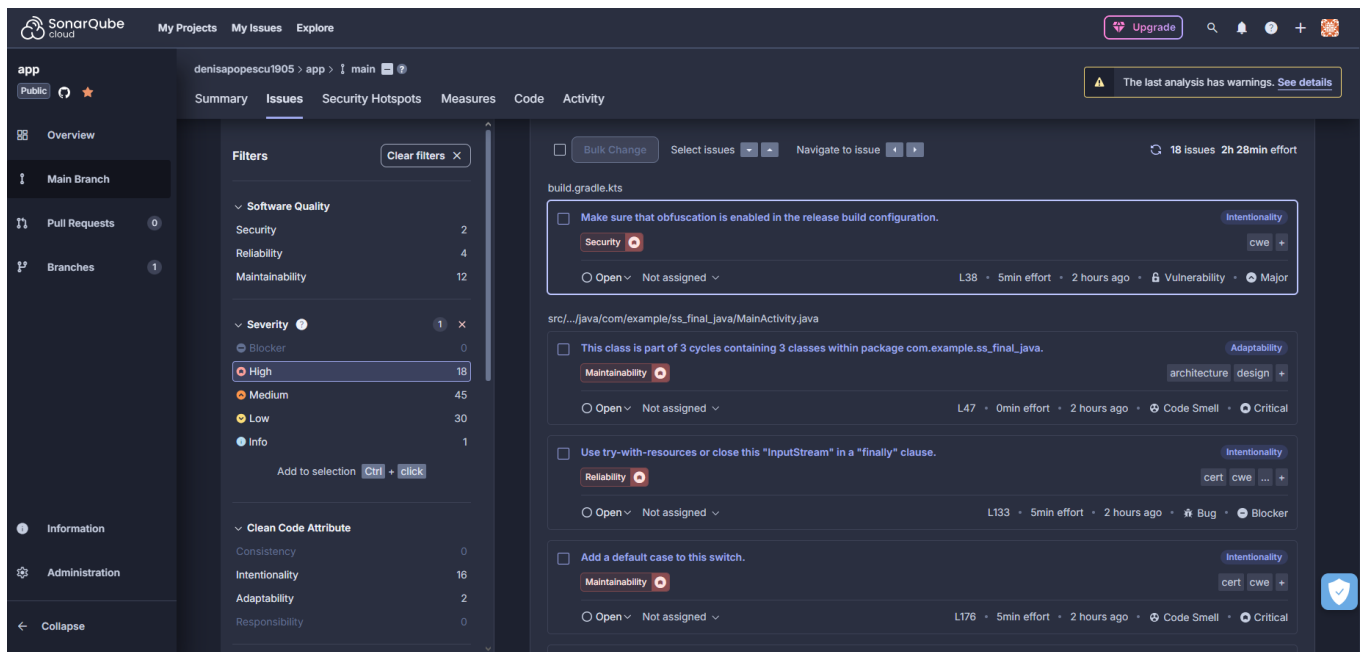


Figure 12: SonarCloud Attack Surface

Overall, there have also been maintainability and reliability concerns in the codebase, some of which have already been addressed. These issues are continuously monitored and re-scanned with every build to ensure ongoing code quality and security improvements.

4.3.1 Fixed Vulnerabilities

Several key vulnerabilities identified during the analysis were addressed to improve the security and reliability of the application.

On one hand, the release build was secured by enabling code obfuscation and shrinking with ProGuard/R8, making reverse engineering significantly harder. Resource leaks caused by unclosed `FileOutputStream` instances were fixed by implementing try-with-resources, ensuring proper closure of system resources.

On the other hand, the critical security issue of disabled hostname verification in SSL/TLS connections was resolved by reinstating proper hostname verification, protecting against Man-in-the-Middle attacks.

Finally, the improper modification of static fields from instance methods was corrected by making the modifying methods static or synchronizing access to guarantee thread safety, preventing potential concurrency issues. These fixes collectively enhance the robustness and security posture of the application.

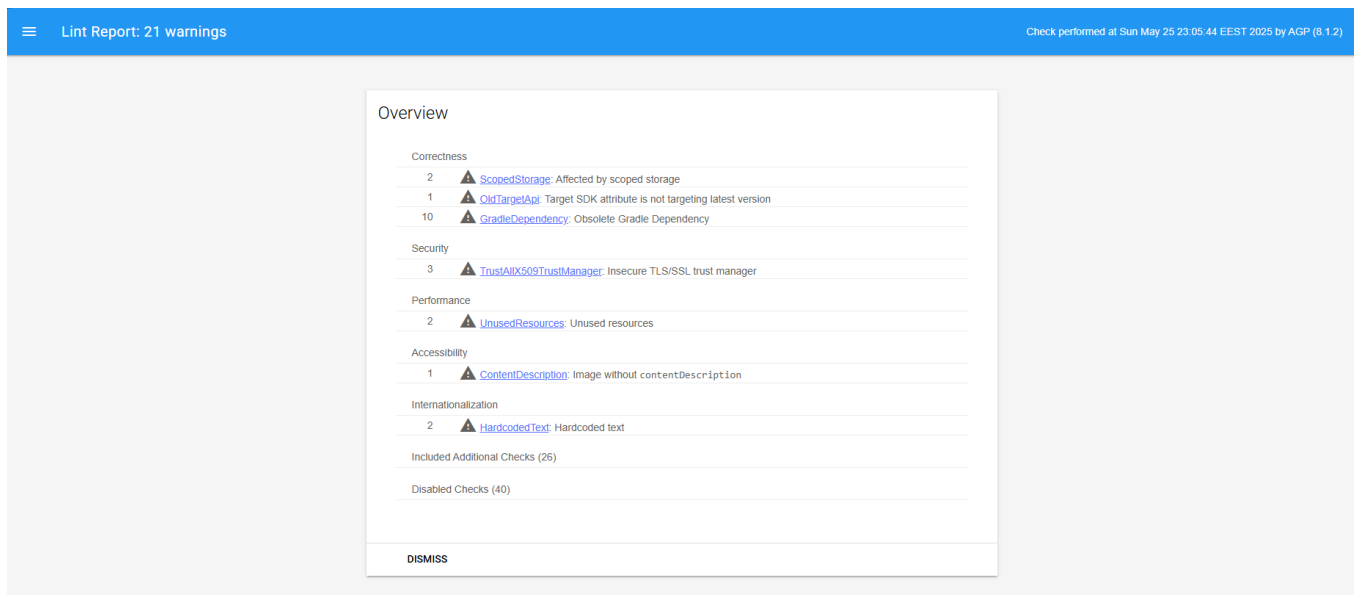


Figure 13: Lint Report

4.3.2 Flask Backend Security Measures and Fixes

In the Flask backend, several potential vulnerabilities and risks were identified and addressed to align the implementation with secure software engineering best practices:

- Insecure Communication:** Initially, the Flask backend was not configured to use secure transport protocols. This was mitigated by using mutual TLS (mTLS) via the `paho-mqtt` client, requiring both server and client certificates for message exchange, effectively eliminating the risk of unauthorized access to MQTT messages.
- Lack of Input Validation:** The backend receives raw image data over MQTT. To prevent malformed or dangerous payloads, exception handling was implemented to verify the image format, catch decoding errors, and reject invalid messages gracefully.
- Excessive Logging of Sensitive Data:** Debug logs that previously printed message contents and file paths were reduced or anonymized to avoid accidental leakage of sensitive data. Logging of base64 payloads or extracted OCR content was scoped and sanitized.
- Missing Security Headers:** The Flask web service was hardened by adding the Flask-Talisman middleware to enforce security headers such as `Strict-Transport-Security`, `X-Frame-Options`, and `Content-Security-Policy`, mitigating browser-based attacks if the HTTP endpoint is ever exposed.
- Unvalidated Third-party Dependencies:** Regular vulnerability scanning of dependencies was introduced using `pip-audit`. Known CVEs in core packages like `setuptools` were identified and fixed by upgrading to secure versions. A GitHub Actions workflow ensures these checks run automatically with every build.
- Unbounded File Writing:** To prevent resource exhaustion attacks or path traversal, the backend was restricted to save incoming images only in a fixed directory (`received_images/`) with sanitized filenames based on timestamps. Proper file

extension enforcement and directory validation were added.

- **Unscanned Code Paths:** Python static analysis was introduced using **Bandit**. Configuration was added to enforce rules for cryptographic misuse, subprocess execution, and insecure function usage. The Bandit HTML report is now automatically generated and uploaded as a CI artifact.

4.3.3 Software Bill of Materials (SBOM) and Dependency Transparency

To further enhance the security posture of the Flask backend and align with modern software supply chain practices, we generated and managed a Software Bill of Materials (SBOM) as part of the development workflow. The SBOM provides a comprehensive inventory of all third-party dependencies used in the backend application, including transitive packages and their exact versions.

The SBOM generation was performed using **pip-audit**, which not only identifies known vulnerabilities (CVEs) in installed packages, but also outputs a structured manifest of dependencies. This information allows for:

- **License Auditing:** Ensuring that all dependencies are compatible with the project's intended license.
- **Vulnerability Management:** Automating the detection of outdated or insecure libraries, such as the CVEs identified in **setuptools**.
- **Supply Chain Transparency:** Providing external reviewers or security teams with a verifiable list of open-source components used at build time.

The SBOM is integrated into the CI/CD pipeline through GitHub Actions, ensuring it is re-generated and re-verified on every build. While the current implementation supports HTML and text outputs, it can be extended to generate standardized CycloneDX or SPDX SBOM formats for integration with broader tooling.

This SBOM-based approach, together with vulnerability scanning and static analysis, helps fulfill industry recommendations such as those outlined by OpenSSF, OWASP SAMM, and NIST SP 800-218 (SSDF).

5 User Documentation

5.1 Overview

The **MQTTImage** Android application enables users to securely capture images and transmit them to a backend server using a secure MQTT connection protected by mutual TLS (mTLS). It supports multiple operation modes, allowing flexible image capture and transmission depending on user needs and network conditions.

5.2 System Requirements

- Android device with at least Android 8 (SDK 26) or higher.
- Network connectivity (Wi-Fi or mobile data) for sending images to the backend.

- **Permissions:** The app requires access to the camera, internet, network state, and device wake-lock to function correctly.

5.3 Permissions Explanation

- **Camera:** To take pictures within the app.
- **Internet:** To connect and communicate with the backend server.
- **Network State:** To monitor network changes and manage connection accordingly.
- **Wake Lock:** To keep the app active during important tasks like uploading images.
- **Receive Boot Completed:** To allow the app to automatically restart after device reboot for continuous operation.

5.4 How to Use

1. **Launching the App:** Open the *MQTTImage* app on your device.
2. **Selecting Mode:** Choose the operational mode using the mode selector:
 - **NONE:** Images are sent immediately after capture.
 - **ON DEMAND:** Images are sent only when you press the “Capture” button or the backend requests it.
 - **PERIODIC:** Images are automatically sent every 30 seconds.
 - **LIVE:** Images are captured and sent continuously every second, providing a live stream.
3. **Capturing Images:** Press the “Capture Image” button to take a picture manually.
4. **Deleting Images:** Use the “Delete Local Images” button to clear any images stored on your device.
5. **Automatic Handling:** The app manages network connectivity and will automatically resend stored images if they couldn’t be sent earlier due to connection loss.

5.5 Connectivity and Security

The app uses secure MQTT connections over mutual TLS to ensure your data is encrypted and only accessible by authorized parties. Certificates are securely loaded, and the app verifies server identity to protect against security threats.

5.6 Troubleshooting

- **No images being sent?** Check your internet connection and app permissions.
- **App not capturing images?** Make sure camera permission is granted.
- **Images delayed?** This might be due to the selected mode or network issues. Try switching modes or reconnecting.

- **App not starting on reboot?** Ensure the app has permission to run at device startup.

5.7 Additional Notes

- The app runs background services to monitor network state and maintain secure connections.
- Images are temporarily stored locally if the network is unavailable and uploaded automatically when connectivity is restored.
- The backend processes received images and extracts text via OCR, enabling automated analysis.

6 Conclusions And Team Contributions

This project successfully developed a secure and reliable Android application for capturing and transmitting images to a backend system via MQTT over mutual TLS. The implementation focused on strong security practices, including encrypted communication, certificate-based authentication, and proper resource management. The backend effectively processes images using OCR, enabling automated extraction of textual data.

Throughout the development lifecycle, rigorous testing, code analysis, and CI/CD automation ensured code quality and maintainability. Identified vulnerabilities were addressed with appropriate mitigations, and the system architecture was designed to handle fluctuating network conditions gracefully.

Future improvements may include expanding functionality, optimizing performance, and extending the security model to support additional use cases and threat scenarios.

6.1 Team Contributions

The contribution data was collected using various Git command line tools. For example, to retrieve the number of commits per author, the command

```
git shortlog -s -n --all
```

was used, which lists authors sorted by the number of commits.

To measure lines added and removed by each author, the following command pipeline was employed:

```
git log --all --pretty=format:"%an"
--numstat | awk '/^[^\\t]+$/ { author=$0;
commits[author]++; next } NF==3 {
added[author]+=$1; removed[author]+=$2 }
END { for (a in added)
printf "%s\\tAdded: %s\\tRemoved: %s\\tCommits: %s\\n",
a, added[a], removed[a], commits[a] }'
```

Table 2 summarizes each team member's contributions based on Git statistics including lines added, lines removed, and number of commits.

Team Member	Lines Added	Lines Removed	Number of Commits
Ioana-Denisa Popescu	3933	1328	58
George-Andrei Cordis	730	160	16

Table 2: Summary of Git contributions per team member

6.2 Default OSSF criticality score result

The `criticality_score` tool ¹⁹ successfully analyzed our GitHub repository <https://github.com/denisapopescu1905/SSProject> and assigned it a default criticality score of **0.20471**. This relatively low score indicates that, while the project has some activity and contributors, it is not yet considered critical within the broader open source ecosystem. Key metrics contributing to this score include the project being recently created (only 2 months old), having 3 contributors, and a modest commit frequency of 1.6 commits per week. Additionally, the repository has no releases, issue activity, or stars, which are all factors that would typically raise the score. This score may increase in the future as the project matures, gains users, and shows more development and maintenance activity.

¹⁹https://github.com/ossf/criticality_score

A Custom SSL Socket Factory Initialization

Listing 4: Custom SSL Socket Factory initialization

```
1 static SSLSocketFactory getSocketFactory(Context context, String
  password) throws Exception {
2     // Add BouncyCastle security provider
3     Security.addProvider(new BouncyCastleProvider());
4     CertificateFactory cf = CertificateFactory.getInstance("X.509
      ");
5
6     // Load CA certificate from raw resources
7     X509Certificate caCert;
8     try (InputStream caIn = context.getResources().
       openRawResource(R.raw.ca)) {
9         caCert = (X509Certificate) cf.generateCertificate(caIn);
10    }
11
12    // Load client certificate from raw resources
13    X509Certificate clientCert;
14    try (InputStream certIn = context.getResources().
      openRawResource(R.raw.client)) {
15        clientCert = (X509Certificate) cf.generateCertificate(
          certIn);
16    }
17
18    // Load and parse client private key (PEM format)
19    InputStream keyIn = context.getResources().openRawResource(R.
      raw.client_key);
20    PEMParser parser = new PEMParser(new InputStreamReader(keyIn)
      );
21    Object keyObj = parser.readObject();
22    parser.close();
23
24    JcaPEMKeyConverter converter = new JcaPEMKeyConverter().
      setProvider("BC");
25    PrivateKey privateKey;
26
27    if (keyObj instanceof PEMEncryptedKeyPair) {
28        // Decrypt encrypted key with password
29        PEMDecryptorProvider decProv = new
          JcePEMDecryptorProviderBuilder().build(password.
            toCharArray());
30        privateKey = converter.getKeyPair(((PEMEncryptedKeyPair)
          keyObj).decryptKeyPair(decProv)).getPrivate();
31    } else if (keyObj instanceof PEMKeyPair) {
32        privateKey = converter.getKeyPair((PEMKeyPair) keyObj).
          getPrivate();
33    } else if (keyObj instanceof org.bouncycastle.asn1.pkcs.
      PrivateKeyInfo) {
34        privateKey = converter.getPrivateKey((org.bouncycastle.
          asn1.pkcs.PrivateKeyInfo) keyObj);
```

```

35     } else {
36         throw new IllegalArgumentException("Unsupported key
           format: " + keyObj.getClass());
37     }
38
39     // Initialize KeyStore with client cert and private key
40     KeyStore keyStore = KeyStore.getInstance("PKCS12");
41     keyStore.load(null, null);
42     keyStore.setKeyEntry("client", privateKey, "".toCharArray(),
           new Certificate[]{clientCert});
43     KeyManagerFactory kmf = KeyManagerFactory.getInstance(
           KeyManagerFactory.getDefaultAlgorithm());
44     kmf.init(keyStore, "".toCharArray());
45
46     // Initialize TrustStore with CA cert
47     KeyStore trustStore = KeyStore.getInstance(KeyStore.
           getDefaultType());
48     trustStore.load(null, null);
49     trustStore.setCertificateEntry("ca", caCert);
50     TrustManagerFactory tmf = TrustManagerFactory.getInstance(
           TrustManagerFactory.getDefaultAlgorithm());
51     tmf.init(trustStore);
52
53     // Create SSLContext with both KeyManagers and TrustManagers
54     SSLContext sslContext = SSLContext.getInstance("TLS");
55     sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(),
           null);
56
57     return sslContext.getSocketFactory();
58 }

```


Vulnerability	Description	Mitigation Strategy	Status
Unobfuscated Release Build in build.gradle	APK is not obfuscated ('isMinifyEnabled = false'), making reverse engineering easy (extraction of hardcoded secrets, API endpoints)	Enable code shrinking and obfuscation with ProGuard or R8 ('isMinifyEnabled = true')	Closed
Circular Dependency	'MainActivity' is part of 3 mutual dependencies, reducing modularity (violates Separation of Concerns principle) and increasing architectural complexity	Refactor classes to remove tight coupling, extract shared logic into separate utility/service classes	Under Analysis
Disabled hostname verification in SSL/TLS	The method <code>setSSLHostnameVerifier((hostname, session) -> true)</code> disables server identity verification, allowing Man-in-the-Middle (MitM) attacks. Detected in <code>MqttHandler.java</code> (line 81).	Enable proper hostname verification by removing the insecure lambda or setting a secure <code>HostnameVerifier</code> implementation.	Open (for System Testing Purpose)
Resource leak: FileOutputStream not closed	<code>FileOutputStream</code> is opened without being closed properly, which may cause resource leaks and reduce reliability. Detected in file saving code.	Use try-with-resources or close the <code>FileOutputStream</code> in a finally block to ensure proper resource management.	Closed
Improper modification of static fields from instance methods	Updating static fields (e.g., <code>currentMode</code>) inside non-static methods can cause concurrency issues and bugs in multi-threaded environments due to unsynchronized access.	Either make the method static when modifying static fields or synchronize access properly; avoid modifying static fields from instance methods without proper thread-safety.	Open

Table 1: Threats identified and their corresponding mitigations