University of Padova

Department of Mathematics "Tullio Levi-Civita"

Master Thesis in Cybersecurity

LLM-Based Triaging of Vulnerabilities in Android Native Libraries

Supervisor
Prof. Eleonora Losiouk
University of Padova

Co-supervisor
PhD Samuele Doria
University of Padova

Master Candidate
Nicola Busato

Student ID
2119291

Academic Year
2024-2025

Abstract

Android applications frequently embed native C/C++ libraries accessed via the Java Native Interface (JNI), introducing memory-safety risks that may compromise the entire application despite the sandbox. Large-scale fuzzing systems such as POIROT generate thousands of crashes for these libraries, but their triage remains manual, slow, and difficult to scale.

To improve the scalability of crash triage for Android native libraries, this thesis quantifies whether an Large Language Model (LLM) equipped with Model Context Protocol (MCP) for retrieving code context, can reliably classify fuzzing-generated crashes and distinguish benign faults from real vulnerabilities.

A structured, automated triage workflow is developed by combining LLM with reverse-engineering tools exposed through the MCP. Crash artefacts generated by POIROT, a state-of-the-art system that automatically synthesises fuzzing harnesses for Android native libraries and produces crash outputs such as stack traces, are enriched with contextual evidence retrieved from Jadx (Java/DEX decompilation) and Ghidra (native decompilation). The LLM analyses each crash following a system-prompt that enforces structured reasoning and a strict JSON schema, leveraging a filtered Java-to-native call graph and a map of crash-relevant native methods to produce a grounded vulnerability assessment including severity, CWE mapping, evidence items, and exploitability indicators.

Across 137 crashes from 80 real-world applications, the system achieves an overall accuracy of 66 %, with a low false-negative rates (3–5 %). When Java-side context is available through a filtered Java Call Graph (JCG), accuracy increases to 77 % and precision more than doubles, confirming the importance of cross-layer information in reducing over-approximation. A detailed case study on TP-LINK's tpCamera reproduces and correctly characterises the real vulnerability later assigned CVE-2023-30273, demonstrating that the system can recover expert-level reasoning patterns using structured evidence.

The findings show that LLM-based crash triage, when grounded through MCP-mediated retrieval of Java and native context, provides a practical and scalable first-line vulnerability assessment mechanism for Android native libraries. While not a substitute for manual auditing, the workflow improves consistency, reduces analyst effort, and offers structured, evidence-driven starting points for further security investigation.

Contents

List of figures

List of tables

List of acronyms

ABI	Application Binary Interface
AI	Artificial Intelligence
API	Application Programming Interface
APK	Android Package
ART	Android Runtime
AWS	Amazon Web Services
CLI	Command-Line Interface
CoT	Chain of Thought
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DEX	Dalvik Executable
EC2	Elastic Compute Cloud
GUI	Graphical User Interface
IPC	Inter-Process Communication
JCG	Java Call Graph
JNI	Java Native Interface
LLM	Large Language Model
MCP	Model Context Protocol
NDK	Native Development Kit
NVD	National Vulnerability Database
PoC	Proof of Concept
RAG	Retrieval Augmented Generation
UID	Unique User ID
VM	Virtual Machine

1 Introduction

Android powers most of the world's smartphones (with around a 75% market share), shaping a vast, heterogeneous ecosystem of devices, vendors, and software variants [1, 2].

At this scale, with a marketplace hosting millions of apps and extensive reuse of the same third-party libraries across thousands of packages.A single vulnerable native component can propagate widely.

Despite Android's multi-layered security model, which includes application sandboxing, permission mediation, SEAndroid/SELinux policy, and verified boot, vulnerabilities persist across apps, frameworks, and vendor components [2, 3]. High-impact examples include memory-safety vulnerabilities in media codecs, that have enabled fully remote, zero-interaction compromise of commercial devices delivered via MMS [4]. Furthermore, empirical evidence indicates that popular apps often embed third-party components with known CVEs and that patch adoption on the app side often lags behind fixes to upstream libraries [5].

Within this landscape, many Android applications embed native C/C++ libraries to achieve low latency, reuse existing code, or access device- and vendor-specific functionality. These libraries are invoked across the JNI boundary, which bridges managed code and native components. While effective for performance, this practice imports the memory-unsafe semantics of C/C++ into an otherwise memory-safe application model, increasing the likelihood of buffer overflows, use-after-free, and related memory-corruption defects. Because native code executes in the same process as the ART, a flaw in the native layer can compromise the entire app and its data, irrespective of the safety of its Java/Kotlin components [5, 6].

A common approach to uncover software defects at scale is fuzzing, an automated dynamic testing technique that feeds programs with large volumes of unexpected, invalid, or mutated inputs to trigger abnormal behaviours. By exercising a wide range of execution paths, fuzzing reveals crashes that may indicate underlying vulnerabilities.

Recent works automate the generation of harnesses and large-scale fuzzing for Android native libraries. Tools such as POIROT can synthesise app-specific harnesses that support bidirectional JNI interactions and then fuzz them at scale, reporting thousands of distinct crashes and confirmed vulnerabilities [7]. However, triaging the resulting crashes, separating benign faults from security-relevant memory errors, remains largely manual.

This thesis investigates whether a tool-grounded Large Language Model (LLM) can reduce manual effort and improve consistency in crash triage for Android native libraries. To this end, it introduces an architecture[1] in which POIROT causes crashes; an LLM-based triager consumes the associated reports and augments them with program context via the Model Context Protocol (MCP), invoking reverse-engineering tools (Jadx for Dalvik/bytecode and manifest, and Ghidra for native disassembly/decompilation) to recover stack evidence, function names, and short decompiled snippets. The triager classifies whether a crash likely indicates a vulnerability, explains the rationale in plain terms, provides a severity estimate anchored to standard taxonomies/scores (CWE/CVSS), and suggests follow-up actions.

[1] Implementation available at `https://github.com/Nicola-01/LLM-Triaging`.

2 Background

This chapter provides the technical background required to contextualise the approach developed in this thesis. It outlines the key concepts underpinning native code execution in Android, the role of the Java Native Interface, and the implications of integrating C/C++ components into modern applications. The chapter also introduces the fuzzing and reverse-engineering techniques that support automated crash discovery and analysis, providing the necessary foundation for understanding the design of the proposed LLM-based triage pipeline.

2.1 Android Native

Android itself relies extensively on native C/C++ libraries within its core architecture. Many system components and services, such as the runtime, media stack, graphics pipeline, and hardware abstraction layer (HAL), are implemented in native code for reasons of performance, portability, and direct hardware access [8]. The framework exposes parts of this functionality to applications through Java/Kotlin Application Programming Interfaces (APIs), enabling managed code to invoke native system services when needed. Figure 2.1 illustrates the Android platform architecture, highlighting how native libraries underpin the runtime, system services, and the HAL.

Beyond the platform itself, developers can extend Android apps with their own native code through the Native Development Kit (NDK).

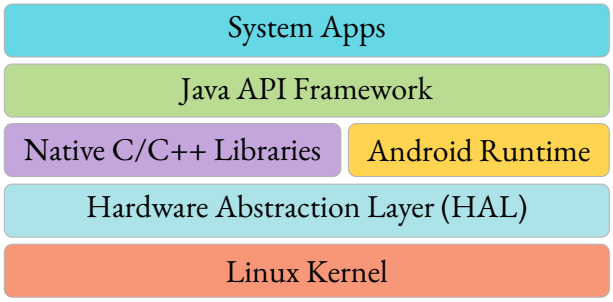


Figure 2.1: Android software stack [8]

The advantages of native development stem from the ability to execute compiled machine code directly on the target CPU, allowing finer-grained optimisation and efficient memory use, important for resource-constrained mobile devices. It also enables hardware acceleration and access to specialised APIs (e.g., GPU, DSP, or sensor interfaces), making it indispensable in areas like augmented or virtual reality, game engines (e.g., Unity, Unreal), and device-specific system utilities.

However, using native code extends the scope of Android applications beyond the safety of the managed runtime. Developers must explicitly handle memory allocation and deallocation, thread synchronisation, and exception propagation, as these are not automatically managed by the Android Runtime (ART). Cross-ABI compatibility, debugging complexity, and maintenance across Android versions further increase the development load. Consequently, the NDK is recommended only when its benefits outweigh these costs [9, 10, 11].

2.1.1 Native Development Kit

The NDK is a collection of tools, headers, and libraries that enable developers to embed C and C++ code within Android applications and interact natively with hardware, sensors, and system APIs [9]. The NDK supports compilation into shared and static libraries that can be packaged inside the Android Package (APK), and offers native interfaces for tasks such as sensor input, asset loading, and more [10].

Developers typically adopt native code for three main reasons:

- **Performance optimization:** achieving low-latency processing in compute-intensive domains such as graphics, signal processing, physics, or cryptography.
- **Library reuse:** integrating existing C/C++ libraries (e.g. cryptography, compression, codecs) to avoid rewriting functionality in Java/Kotlin.

- **Hardware or vendor-specific access:** interacting directly with low-level or proprietary APIs (e.g. custom sensors, specialized accelerators) not exposed in the Java framework.

However, using the NDK comes with tradeoffs. Native development adds complexity in build configuration, cross-ABI support, debugging, and maintenance across Android versions. Not all Android APIs are directly available through the NDK, so bridging via Java Native Interface (JNI) is often required for broad framework functionality [10].

2.1.2 Java Native Interface

The JNI is a native programming interface that enables Java or Kotlin code running in a virtual machine to interoperate with libraries and applications written in C/C++ [12]. It allows managed code to call into native code, and for native code to call back into the Virtual Machine (VM), manipulate Java objects, throw exceptions, and more. The JNI is designed to impose no restrictions on the implementation of the VM, thereby preserving binary compatibility across vendors [12].

Listing 2.1 shows a Java class declaring a `native` method, while Listing 2.2 gives the corresponding C implementation on the native side.

```java
public class foo {
    private native double bar(int i, String s);
    static {
        System.loadLibrary("native-lib");
    }
}
```

Listing 2.1: Java class declaring a `native` method

```c
jdouble Java_pkg_foo_bar(JNIEnv *env,          // ptr to JNI interface
                         jobject obj,          // "this" pointer
                         jint i, jstring s) {  // first and second parameter
    return 0.0; /* Method implementation */
}
```

Listing 2.2: Native implementation of `bar`
The native function name `Java_pkg_foo_bar` follows the JNI name-mangling convention: it is composed of the prefix `Java_`, the fully qualified class name (`pkg.foo`, with dots replaced by underscores), and the Java method name `bar`, thus uniquely identifying the native implementation of `foo::bar(int,String)`.

2.1.3 Vulnerabilities and Impact on the Android Sandbox

Native memory-safety bugs remain a dominant cause of serious compromise on Android. Although the platform's sandbox assigns each app a distinct Unique User ID (UID) and further constrains it with SELinux policies, a native component compromised inside that boundary still runs with the app's privileges and can expose secrets or capabilities already permitted (e.g., tokens, stored content, keys) [13, 14]. In practice, this turns local corruption into high-impact data access or control-flow hijacking even without crossing process boundaries.

Memory-safety defects in native libraries, such as buffer overflows, out-of-bounds reads/writes, use-after-free, and double free, belong to well-known Common Weakness Enumeration (CWE) families and frequently appear in Common Vulnerabilities and Exposures (CVE) records. On Android, these errors may cause crashes or, in more severe cases, permit code execution or data tampering depending on mitigations and exploitability assumptions [15].

Attackers often amplify impact by chaining flaws across interfaces. Many system services are native and reachable over Binder; memory corruption in such a service can be triggered via Inter-Process Communication (IPC) by a malicious client to bypass app-level limits [16]. In other scenarios, native code may corrupt memory shared with the managed runtime (e.g., altering JNI or class metadata) by abusing memory-protection system calls, undermining Java-level safety invariants and enabling code injection or VM subversion [17].

2.2 Fuzzing

Fuzzing is an automated testing technique that repeatedly executes a target program with many inputs, either randomly generated or systematically derived, to expose defects such as crashes, assertion failures, or memory-safety violations. Modern fuzzers prioritise inputs that increase code coverage (e.g., edge or block coverage), and couple execution with hardening oracles (e.g., sanitizers) to turn latent memory errors into reliable, actionable signals [18, 19].

A fuzzer can be classified along three ways:

- **Input production:** *generation-based* where inputs are produced from a model or specification and *mutation-based* where inputs are produced by mutating existing seeds. Coverage-guided mutation fuzzers (e.g., AFL++, libfuzzer) iteratively mutate seeds that increase coverage.
- **Input knowledge:** *structured*, grammar- or model-based, aware of input format/protocol, or *unstructured*, no knowledge of structure; "dumb" or purely random.

- **Program knowledge:** *black-box*: no visibility into internals, *grey-box*: lightweight instrumentation, e.g., edge coverage, and *white-box*: constraint solving/symbolic execution to systematically steer execution, as in SAGE [19].

Fuzzing is primarily employed to discover security-relevant defects in safety- or security-critical software. Its strength lies in *demonstrating the presence* of bugs via concrete, reproducible inputs. Proving correctness for all inputs requires a formal specification and formal methods; fuzzing complements, rather than replaces, such approaches [18].

On Android, fuzzing native components benefits from memory error detectors such as Hardware-assisted AddressSanitizer (HWASan) and AddressSanitizer (ASan). These sanitizers instrument code to detect out-of-bounds accesses, use-after-free, and related violations during fuzzing, producing precise diagnostics that accelerate crash triage [20, 21, 22].

Two widely used coverage-guided engines are American Fuzzy Lop++ (AFL++) (a fork and extension of AFL with a rich ecosystem of mutators and instrumentation options) and LLVM In-Process Coverage-Guided Fuzzer (libFuzzer). Both have been successfully applied to native libraries, including on Android, to elicit high-quality crashes and to maximise coverage under realistic budgets [23, 24].

2.3 Large Language Model

LLMs are deep neural networks based on the Transformer architecture, which replaces recurrence/convolution with stacked self-attention and feed-forward layers to model long-range dependencies efficiently [25]. Today's LLMs are pre-trained on large corpora via next-token prediction and then adapted to follow user prompts.

2.3.1 Limitations and Hallucinations

Because LLMs predict tokens rather than verify facts, they can generate confident but incorrect content ("hallucinations"). Surveys and recent studies document multiple forms (e.g., intrinsic or confabulated outputs) and show that even state-of-the-art models may produce fluent fabrications [26, 27]. In security-oriented workflows such as vulnerability triage, this can manifest as invented CVE identifiers, non-existent APIs or packages, fabricated proof-of-concept details, or misattributed stack traces, errors that can mislead severity assessment or waste analyst time. Empirical work on code-generation further highlights "package hallucinations", where mod-

els recommend non-existent libraries, creating supply-chain risk if an attacker later publishes a malicious package with that name [28].

2.4 Model Context Protocol

MCP is an open protocol, introduced by Anthropic (creator of Claude) in November 2024, that standardises how Artificial Intelligence (AI) applications obtain and manage external context via a client–server architecture. An *MCP host* (the AI application) connects to one or more *MCP servers* through dedicated *MCP clients*, enabling the host to discover and use tools, read resources, and apply prompts exposed by servers. The protocol separates *data layer* from a *transport layer* (e.g., stdio for local, streamable HTTP for remote), so the same message semantics work across local and remote deployments [29, 30, 31]. In brief, servers contribute context and actions; clients manage capability negotiation and exchanges; the host orchestrates everything in the user-facing application [29, 32, 33]. Figure 2.2 provides a high-level view of this host–client–server pattern and example integrations.

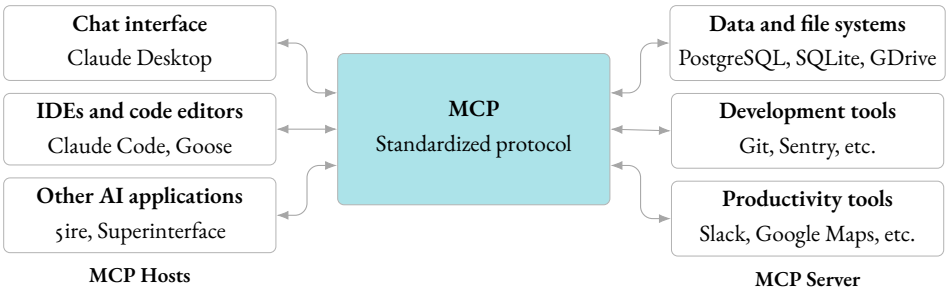


Figure 2.2: High-level MCP architecture. The MCP host (e.g., chat interface or IDE) coordinates one or more MCP clients, each connected to an MCP server that exposes tools, resources, and prompts [30].

2.4.1 MCP Host

The MCP *host* is the application the user directly interacts with (e.g., a chat UI or an IDE extension). It manages the overall user experience, session lifecycle, permissions, and tool exposure, and it instantiates one or more MCP clients to talk to specific servers. In practice, the host governs policy (e.g., which servers are allowed, rate limits) and mediates the user-facing rendering of server responses [29]. Figure 2.3 situates the host on the left, coordinating multiple client–server links while preserving a single conversational surface for the user.

2.4.2 MCP Client

An MCP *client* is a protocol-level component created by the host to communicate with exactly one MCP *server*. Each client maintains a transport (stdio or HTTP/Web Socket), negotiates capabilities, and exposes the server's tools/resources/prompts to the host. This indirection lets the host manage several servers in parallel without coupling UI logic to any server implementation details [33]. In Figure 2.3, clients appear as the "middle" connectors: they translate host intents (e.g., "run tool T") into API call and relay the server's results back to the host.

2.4.3 MCP Server

An MCP *server* wraps external systems (e.g. files, databases, reverse-engineering tools) and presents them as protocol objects. It exposes *tools*, which expose invokable functions with JSON Schemas; *resources*, which provide readable artefacts; and *prompts*, which offer reusable task templates. Servers publish a capability manifest so that clients can discover what is available and how to access it. The MCP *server* provides tools that enable LLMs to perform actions, carry out deterministic computations and interact with external services. This design decouples model/UX from system integrations and enables reuse across different hosts and models [34]. In Figure 2.3, servers sit on the right, each encapsulating one integration boundary (e.g., Jadx, Ghidra, a code search index).

2.4.4 Interaction

Figure 2.3 illustrates a typical flow.

1. The host initialises and lists available servers (or lets the user attach new ones).
2. For each server, the host spawns a client that performs capability discovery (tools, resources, prompts and their schemas).
3. When the user or an agentic workflow triggers an action, the host selects an appropriate tool and routes a request through the corresponding client to the target server.
4. The server executes the tool (e.g., get app manifest with Jadx, decompile a function with Ghidra) and returns structured results to the client.
5. The client relays the structured results to the host; the host provides this context to the LLM, and may chain further tool calls.
6. The LLM synthesises a user-facing response using the returned context; the host renders this reply in the UI (and records any artefacts/evidence links).

7. The host may persist state and expose follow-up actions (e.g., re-run with new arguments, open related resources), maintaining a single conversational surface.

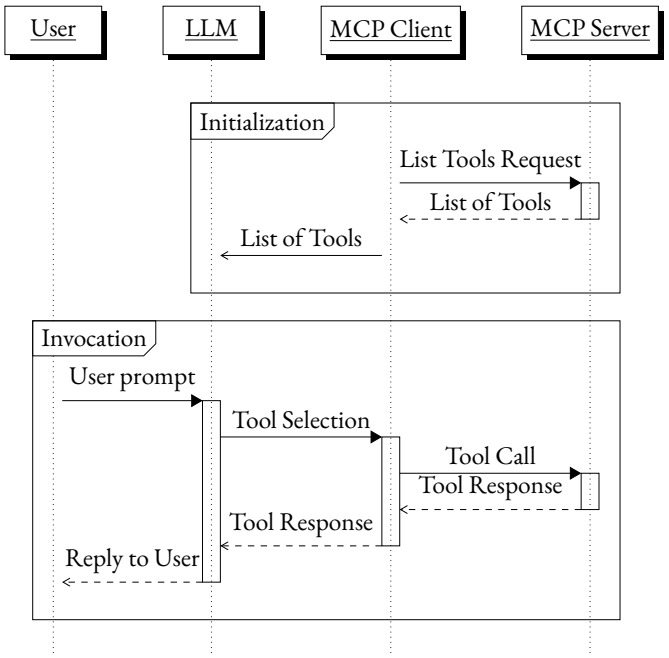


Figure 2.3: MCP interaction: capability discovery, tool selection, invocation, and reply.

2.5 Vulnerability Scoring Systems

In modern security analysis, vulnerabilities are not only discovered but must also be described, categorised, and prioritised in a consistent manner. To support this process, widely adopted standardised frameworks provide identifiers, taxonomies, and severity metrics that allow tools, researchers, and vendors to communicate about security issues in a uniform way.

2.5.1 Common Weakness Enumeration (CWE)

The CWE is a community-curated taxonomy of common software and hardware weakness types (e.g., buffer overflow, use-after-free). It provides standardised identifiers and structured descriptions to support detection, prevention, and education across the secure development lifecycle [35]. Unlike CVE (which tracks specific *instances*), CWE captures *classes* of underlying faults, enabling aggregation and trend analysis at the root-cause level [36].

2.5.2 Common Vulnerabilities and Exposures (CVE)

The CVE Programme assigns unique identifiers to publicly disclosed vulnerabilities, providing a single canonical reference for coordination across vendors, researchers, and users. A CVE entry names the specific issue (product, version, and vulnerability description) and links to public references; other databases may enrich it with severity and technical details [37]. In practice, national or vendor repositories (e.g., National Vulnerability Database (NVD)) attach CVSS scores and map each CVE to one or more CWE categories for analytics and prioritisation [38, 39].

2.5.3 Common Vulnerability Scoring System (CVSS)

The Common Vulnerability Scoring System (CVSS) is an open, vendor-neutral framework for describing the intrinsic characteristics and severity of a vulnerability through a set of metrics that yield a numerical score and qualitative severity band. CVSS communicates *severity*, not overall *risk*; operational risk assessment should incorporate asset context, threat activity, and environmental factors beyond the Base score [40].

2.5.4 Relation

In triage and remediation workflows, CVE entries identify concrete vulnerabilities; each entry can be mapped to CWE identifiers that explain the underlying weakness type; CVSS scores (often provided by downstream databases) express the vulnerability's technical severity. Together, CWE→CVE→CVSS forms a pipeline from root-cause taxonomy to instance tracking and severity quantification [38].

2.6 Vulnerability Triage

Vulnerability triage is the process of assessing, prioritising, and deciding on the appropriate response for a set of identified software vulnerabilities [41]. In practice, this means evaluating each potential issue to determine if it represents a real security risk, estimating its impact and exploitability, and selecting which findings deserve immediate attention or remediation.

2.7 Reverse engineering

Reverse engineering is the analysis of binaries or bytecode to recover design and implementation information-disassembly, control/data flows, and higher-level structure, when source or specifications are incomplete or unavailable [42, 43]. In practice, reverse engineering workflows combine a disassembler/decompiler with cross-reference search, symbol inspection, and targeted decompilation to produce evidence suitable for auditing and triage.

2.7.1 Ghidra

Ghidra is an open-source software reverse-engineering framework developed by the National Security Agency (NSA). It offers disassembly, decompilation, graphing and scripting capabilities across numerous architectures and binary formats, and is broadly adopted within vulnerability research and malware analysis communities [44]. In this thesis, it is used to enable the LLM to inspect native binaries: once an ELF library is imported, Ghidra can generate control-flow graphs, symbol tables, cross-references, and decompiled C code. These artefacts are then exposed via the MCP server, allowing the pipeline to query functions, addresses, call-graphs and other fine-grained details necessary for crash triage.

2.7.2 Jadx

Jadx is a decompiler that converts Android DEX and APK files into Java source code, offering both Command-Line Interface (CLI) and Graphical User Interface (GUI) front-ends [45]. In the pipeline, Jadx fulfills two key roles: first, it recovers application metadata (such as the manifest, SDK version, permissions and package structure); second, it decompiles the Java portion of the application, enabling analysis of Java-to-JNI transitions.

3 Related Work

This thesis project addresses a timely and relevant topic: the use of LLMs for automated vulnerability classification, focusing on Android native libraries. While recent research suggests that LLMs can support parts of the vulnerability analysis workflow by leveraging advanced natural language and code understanding, empirical evidence indicates that their effectiveness is strongly dependent on the quality and completeness of the context available to the model.

The need for scalable triage of crashes, where manual analysis is often slow, labour-intensive, and difficult to apply consistently at scale, has motivated the exploration of LLMs as a complementary solution. These models can assist, or partially replace, manual inspection by rapidly interpreting crash reports, extracting relevant execution details, and supporting the identification of potential vulnerability patterns.

3.1 Lack of Context in LLMs

Several studies have shown that LLMs often struggle when analysing security-critical code, especially when key semantic or structural cues are missing. For example, Basic and Giaretta [46] report that current models frequently misclassify vulnerabilities when the prompt lacks essential information about the execution environment or the underlying fault mechanism. These limitations suggest that LLMs rely on superficial patterns when deeper program semantics are inaccessible. In this context, enabling the model to autonomously explore the codebase through the MCP framework is a promising direction: allowing the LLM to retrieve files, inspect call

paths, and gather execution context can mitigate the shortcomings of prompt-only approaches and support more reliable crash interpretation.

3.2 LLM-Assisted Fuzzing

Similar limitations of LLMs have been observed in the fuzzing domain. Jiang et al. [47] show that LLM-assisted fuzzing often fails when models must reason over incomplete or excessively long contexts, leading to inaccurate interpretation of bug-detection signals and weakened semantic understanding. Their findings indicate that providing large amounts of raw code or documentation is ineffective: long, unstructured prompts reduce the model's ability to track dependencies and increase the likelihood of hallucinations or incorrect inferences.

These observations motivate a more selective strategy for context provision. Instead of providing the entire codebase, which has been shown to reduce accuracy, allowing the LLM to retrieve only relevant fragments on demand via the MCP constrains the reasoning space to focused, semantically meaningful evidence. This mitigates the long-context limitations highlighted by Jiang et al. and supports more stable and interpretable analysis workflows.

Liu et al. propose PromeFuzz[48], a framework that augments LLMs with structured information extracted from the target application. Rather than relying solely on function signatures, PromeFuzz builds a knowledge base from code metadata, documentation, and usage correlations, retrieved via Retrieval Augmented Generation (RAG) at inference time, while a dedicated sanitizer filters hallucinated or semantically invalid outputs. This combination of structured context and validation substantially improves code coverage and crash-detection accuracy, showing that LLMs become more reliable when grounded in verifiable program artefacts instead of raw prompt text.

3.3 Vulnerability Triage

Several works explore LLMs for vulnerability triage. CASEY [49] uses contextual information from the NVD to automate CWE and severity classification, while Akuthota et al. [50] demonstrate similar trends in vulnerability detection and monitoring. A broader evaluation by Yin et al. [51] shows that although LLMs underperform in pure detection tasks, their accuracy improves substantially when supplied with additional contextual information such as file names, commit messages, or CVE descriptions. These results further justify the use of MCP to supply structured context directly from the codebase.

Finally, work on automated vulnerability repair reinforces the need for external feedback. VRpilot [52] shows that reasoning steps and validation signals significantly improve repair quality, while LProtector [53] combines GPT-4o with RAG to enhance binary vulnerability detection. Both studies illustrate that LLMs perform best when supported by external signals and structured knowledge.

Overall, the literature indicates that LLMs alone are insufficient for accurate vulnerability classification or crash triage. Their performance improves markedly when complemented by structured context, execution information, and auxiliary analysis tools. These findings motivate the integration of MCP, enabling the LLM to ground its decisions in concrete code evidence rather than prompt-only heuristics.

4 Preliminaries

This thesis relies on crash data generated by fuzzing Android native libraries embedded in APKs via the JNI. To obtain these data, *POIROT*, an automated framework for app-specific harness synthesis and large-scale fuzzing of native libraries on Android [7], was employed. Reproducibility is ensured by adoption of the authors' artifact (Docker-based environment and emulator setup)[1]. POIROT-generated crash artefacts provide the input to the LLM-based triage system.

4.1 Data generation via POIROT

POIROT follows a three-stage process.

I. **Static analysis.** The framework extracts Dalvik Executable (DEX) bytecode and native libraries from each APK and runs two scalable analyses over the Java side: a *call-sequence analysis* to recover app-specific invocation sequences of native APIs, and an *argument analysis* to infer lightweight constraints for parameters (e.g., array-length relations, file-path uses).
II. **Harness synthesis.** From those analyses, POIROT synthesises per-function drivers that (a) construct a minimal ART-backed runtime with a genuine `JNIEnv` to support bidirec-

[1] https://github.com/HexHive/droidot

tional callbacks and (b) encode the recovered call sequence and parameter constraints.

III. **Fuzzing and crash detection.** Harnesses are executed under greybox fuzzing (AFL++ with Frida-based instrumentation), collecting coverage and detecting memory-safety faults. Crashes are deduplicated and recorded together with backtraces and minimal reproducers; allocator/sanitiser diagnostics (e.g., Scudo signals, ASan/HWASan when available) assist categorisation.

4.2 Dataset produced for this thesis

The dataset used for this triager consists of deduplicated *crash reports* emitted by POIROT, while fuzzing native C/C++ code reachable from each APK through the JNI. For each crash, the stack trace is retained. This artifact is cross-referenced with triage based on LLM to reconstruct and analyze the path and provide an evidence-based assessment, by retrieving the necessary information.

Listing 4.1 shows two representative crashes produced when fuzzing the method `Java_com_tplink_skylight_common_jni_MP4Encoder_packVideo` in the application `com.tplink.skylight`. In both cases, the top frames reveal an allocator abort in Scudo (`scudo::reportInvalidChunkState` → `scudo::Allocator…::deallocate`) followed by the target media routines (`mp4_write_one_h264` / `mp4_write_one_jpeg`) and the JNI bridge function (`Java_com_tplink_skylight_common_jni_MP4Encoder_packVideo`). Operationally, this pattern indicates that the runtime detected an invalid deallocation (e.g., double/invalid free) during processing of fuzzer-supplied data, and surfaced it as an immediate `abort`.

Each POIROT crash record stores the full stack trace (as shown in Listing 4.1), from which it is possible to extract the native function name, the JNI entry point, and the likely cause of the crash.

Each method crash is stored in its own file within a unique directory named with the corresponding method. The *Triage* section[2] of the GitHub README provides an example of the directory structure used for a crash.

[2] `https://github.com/HexHive/droidot/#triage`

```
################ CRASH NR 0 ######################
abort
scudo::die
scudo::ScopedErrorReport::~ScopedErrorReport
scudo::reportInvalidChunkState
scudo::Allocator<scudo::AndroidConfig, &scudo_malloc_postinit>::deallocate
mp4_write_one_h264
Java_com_tplink_skylight_common_jni_MP4Encoder_packVideo
fuzz_one_input
main
################################################
################ CRASH NR 1 ######################
abort
scudo::die
scudo::ScopedErrorReport::~ScopedErrorReport
scudo::reportInvalidChunkState
scudo::Allocator<scudo::AndroidConfig, &scudo_malloc_postinit>::deallocate
mp4_write_one_jpeg
Java_com_tplink_skylight_common_jni_MP4Encoder_packVideo
fuzz_one_input
main
################################################
```

Listing 4.1: folder2backtraces.txt file, created as the output of POIROT, for the apk `com.tplink.skylight` in the function `Java_com_tplink_skylight_common_jni_MP4Encoder_packVideo`

Environment

The approach was initially validated locally by running POIROT in Docker on a workstation and executing targets on a physical Android device (Google Pixel 9). Since the number of APKs to process exceeded the capacity of a single device, the data-generation phase was subsequently migrated to a cloud host.

Concretely, Amazon Web Services (AWS) Elastic Compute Cloud (EC2) was used with an ARM-based Graviton instance to match the target `ARM64-v8a` ABI of the native libraries. The host was a `c6g.metal` machine (64 vCPU, 128 GB RAM), which provides an ARM (AArch64) host and hardware virtualisation via KVM. This configuration allows the Android

emulator to run ARM system images directly on an ARM host, avoiding cross-ISA translation and enabling faithful execution.

POIROT's analyses and harness synthesis ran inside the provided Docker environment; fuzzing then executed on the emulator farm on the `c6g.metal` host. The setup produced the same crash artefacts as the local pilot (stack traces, reproducer inputs, managed–native call sequences), but with larger scale.

5 Design

This chapter presents the conceptual design of the automated crash–triage system. The goal is to formalise the architecture, information flow, and reasoning model that underpin the use of LLMs for vulnerability triage of crashes in Android native libraries.

5.1 Data Models

A central element in the design of the crash–triage system is the definition of a set of structured data models that organise all information exchanged between components. Since the LLM-based analysis relies on precise, machine-readable context, every stage of the workflow, from parsing POIROT outputs, to extracting application metadata, to producing the final vulnerability assessment, depends on well-defined representations. These models constrain how information is encoded, ensure consistency across analyses, and guide the LLM by enforcing a predictable input and output format.

In this thesis project, all relevant information is captured through a set of custom data models, `CrashSummary`, `AppMetadata`, and `VulnResult`. Their standardised structure guarantees that the `CrashSummary` provided to the LLM have always a consistent structure, divided into clearly defined fields that are explicitly explained in the system prompt and easily interpretable by the model. Likewise, the `AppMetadata` and `VulnResult` models allow the system to request a precise and schema-constrained output from the LLM, which must adhere to the predefined structure.

Instead of asking the model directly whether the app is vulnerable and why, the system instructs the LLM to populate a fixed JSON template containing all relevant fields, with strict type and content definitions, ensuring that output is both machine- and human-readable. This approach may improve LLM robustness and reduce variability in classification tasks, because schema-based prompting helps the model focus on the attributes relevant for the decision while avoiding drifting into irrelevant details.

Moreover, enforcing a fixed structure simplifies downstream processing: outputs can be exported as well-formed `JSON` files and automatically queried or aggregated across multiple applications. For example, after a full triage run, external utilities can group results by vulnerability verdict, severity level, or associated CWE identifiers, inspect all low-confidence cases, or generate visual summaries such as the ratio of vulnerable versus benign crashes across Android versions.

The crash report may also include a possible Proof of Concept (PoC), meaning that structured outputs can additionally support automated testing: external tools can more easily extract the commands suggested by the LLM and attempt to execute them for exploit verification. This capability is not guaranteed to work reliably in all cases, but a standardised output format makes such integration technically feasible and simplifies the development of external validation utilities.

The structured design therefore relies on a set of core data models:

- **CrashSummary** Represents a single crash extracted from the analysis report.
  It encapsulates key fields such as the native stack frames, JNI bridge method, a reconstructed Java Call Graph (JCG), the set of native libraries involved in the stack trace, among other fields..
- **AppMetadata** Aggregates APK-level information extracted from Jadx, including the application name, package name, version code and version name, and the minimum and target SDK levels.
- **VulnResult** Represents the structured output of the LLM for a single crash.
  It exposes information such as the chain of thought, vulnerability verdict, CWE identifiers, severity level and an optional exploitability assessment via the `Exploit` object, as well as other contextual information.
- **Exploit** Describes the exploitability of a suspected vulnerability.
  It contains key fields such as the exploitability rating (unknown / theoretical / practical), the required environmental prerequisites, and Proof of Concept commands, alongside additional optional elements.

Additional structures are:

- **EvidenceItem** stores an piece of evidence used by the LLM to justify its reasoning, such as a function name, memory address, short decompiled snippet, and an associated explanatory note.
- **Statistics** collects metrics on model usage (tokens, requests, wall-clock time), enabling performance profiling and cost monitoring during large-scale triage.

Some of these data structures will be examined in more detail throughout this chapter.

Overall, this strict modelling architecture ensures resilience against malformed or incomplete LLM outputs and provides a deterministic structure for downstream processing, evaluation, and reproducibility.

5.2 Prompt Design

A central component of the system design is the structure of the prompts used to guide the LLM. Because the triage procedure requires multi-step reasoning, explicit interaction with MCP tools, and the generation of JSON outputs, the quality and organisation of the prompts directly determine the reliability of the analysis. Well-designed prompts reduce ambiguity and also provide a stable decision-making model, ensuring consistent behaviour across different applications and crashes.

5.2.1 Vulnerability Triage Prompt

The core of the system is the vulnerability triage prompt, which governs how the LLM analyses each crash and produces its final assessment. The prompt employs *meta prompting*[54]: the system prompt defines the agent's role, the overall reasoning procedure, the rules for interacting with MCP tools, and the exact JSON structure that the model must produce. This provides the model with a stable cognitive frame and constrains its behaviour.

In addition, the prompt enforces an implicit form of *Chain-of-Thought* reasoning[55]. Although the internal chain of thought is not exposed in the output, the model multi-step analytical path—starting from the crashing frame, performing backward reasoning, checking reachability, and validating assumptions, before emitting the final classification.

The full system prompt used in the implementation is reported in Appendix A.

5.2.2 Structure of the Prompt

The prompt used for vulnerability triage is organised into a set of structured components that collectively define the agent's behaviour, analysis procedure, and output constraints. Rather than providing ad-hoc instructions, the prompt specifies a complete operational framework that the LLM must follow when examining each crash. Its structure can be understood through four conceptual layers.

1. Role and behavioural specification. The prompt begins by defining the role of the agent, its objectives, and the scope of the analysis. It instructs the model to behave as a vulnerability analyst with access to static-analysis tools, explicitly prohibiting unsupported assumptions, unverifiable statements, and speculative reasoning. Clear behavioural rules constrain how the agent must reason about the crash, how it should handle missing information, and how it should justify its conclusions.

Listing 5.1 shows the corresponding section of the prompt.

```
You are a **senior mobile reverse-engineering & security engineer**.
You will receive one CrashEntry at a time from a JNI-fuzzing triage pipeline.
Your task is to decide whether the crash is LIKELY caused by a genuine code
    vulnerability
(memory safety, logic bug, or exploitable condition) or NOT.
Return ONLY a single JSON object that strictly follows the schema below.
```

Listing 5.1: Role and behavioural specification in the vulnerability-triage prompt.

2. Interpretation of inputs. The prompt describes in detail the meaning and expected use of the inputs provided for each crash: the native stack trace, the JNI bridge method, the JCG, and the list of relevant libraries. The model is instructed to integrate these artefacts into a coherent execution path. In particular, it must reconstruct the control flow starting from the point of failure and reason backwards to identify the root cause, assessing whether the crash is plausibly reachable and whether user-controlled data may influence the faulting operation.

Listing 5.2 reports the exact instructions included in the prompt.

```
You will receive the following fields:
- process_termination
- stack_trace
- java_callgraph
```

```
- app_native_function
- jni_bridge_method
- fuzz_harness_entry
- program_entry
- relevant libraries and their JNI methods

You must integrate all fields into a coherent reconstruction of the failing
    execution path.
```

Listing 5.2: Input interpretation rules provided to the LLM.

3. Tool-use protocol. A dedicated section of the prompt specifies how the agent should interact with external tools exposed through the MCP. It explains some tool calls, what information can provide, and how retrieved evidence should influence the ongoing analysis.

Listing 5.3 shows the relevant portion of the prompt.

```
You have **Jadx MCP** and **Ghidra MCP** and MUST use them proactively.

Exploration Rules (excerpt):
1. Resolve thunks/imports:
   - If the crash is in a wrapper, inspect the caller via `LibMap` and
    decompile it.
2. Cross-library search:
   - If a symbol is missing in one `.so`, check exports in other libraries
    (...).
3. JNI root analysis:
   - Always decompile the App Native Function (JNI entry point).
4. Java context:
   - Inspect the `jni_bridge_method` in Jadx to check argument validation.

Steps for each crash (simplified):
1. Identify the first app-level frame below allocators/sanitizers.
2. GHIDRA MCP: Decompile the function at that frame.
3. Backward data-flow: Trace arguments backwards; recurse into callers until
    JNI or validation.
4. JNI/Java analysis: Check how Java constructs arguments and whether they are
     attacker-controlled.
```

```
5. Function-pointer resolution: Resolve indirect calls; if missing, explicitly
    state uncertainty.
```

Listing 5.3: Tool-use protocol and mandatory MCP exploration steps.

4. Structured output requirements. The final part of the prompt imposes strict output constraints. The model must produce a JSON object consistent with a predefined schema, filling fields such as the vulnerability verdict, confidence score, severity, CWE identifiers, and other. Each field is accompanied by explicit instructions describing what constitutes a valid value and how the agent should justify its decisions.

Listing 5.4 reports the schema-oriented instructions.

```
Return a JSON object with:
- `chain_of_thought`: strings. Write a step-by-step internal monologue BEFORE
    classifying.
- `is_vulnerable`: boolean. True if the crash is a vulnerability, false
    otherwise
- ...

## 6a. Exploit field requirements (only when is_vulnerable=true)
When a crash is classified as a real vulnerability:
1. You MUST provide an `exploit` object with concrete, realistic details.
2. The `exploit_pipeline` MUST describe, in 3-5 ordered steps, ...
3. `poc_commands` MUST include at least one actionable Proof-of-Concept
    command..

- `exploit`: null OR an object with:
    - ...

All fields must strictly adhere to the predefined schema.
```

Listing 5.4: Structured output and JSON-schema constraints.

Overall, this structured prompt acts as a complete analytical workflow: it defines how the LLM interprets inputs, how it retrieves additional evidence, how it reasons about the execution path, and how it produces a vulnerability assessment.

5.3 Pipeline Architecture

The triage pipeline is designed as a structured, evidence-driven reasoning process. Its objective is to enrich the raw crash artefacts produced by POIROT with code-level context, enabling the LLM to formulate grounded judgements.

Figure 5.1 provides an overview of the full workflow, which integrates: (i) POIROT's fuzzing output, (ii) static-analysis backends exposed via MCPs, and (iii) the LLM configured with a strict prompt and output schema.

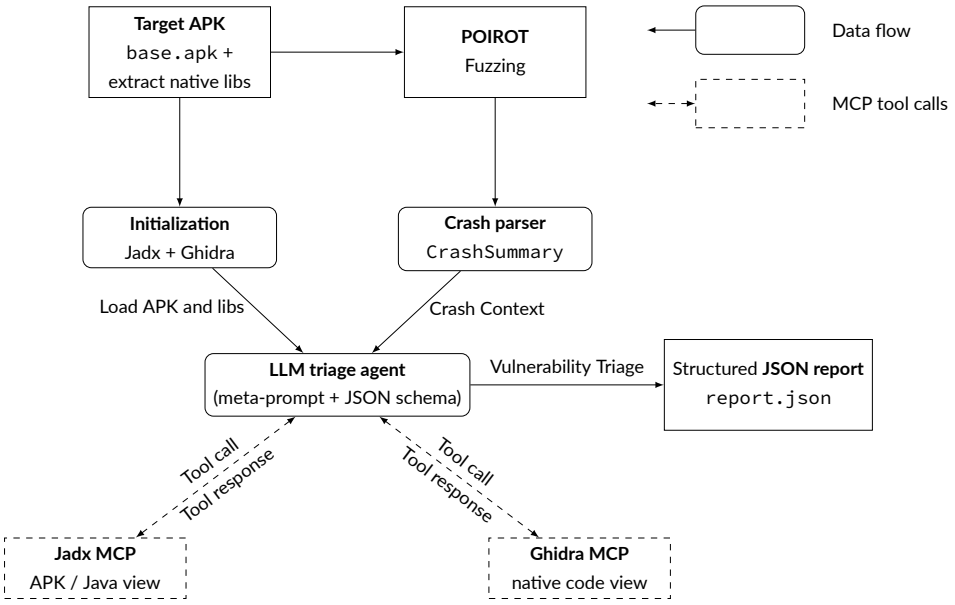


Figure 5.1: Overview of the triage pipeline, combining POIROT fuzzing, crash analysis, and MCP-based inspections.

The pipeline consists of the following stages.

5.3.1 Crash Report Parsing

At the start of the pipeline, the tool retrieves the `folder2backtraces.txt` files (Listing 4.1) generated by POIROT. These raw crash reports contain the native stack traces, from which the process termination cause and the JNI entry point can be extracted.

For each crash, the system constructs a corresponding `CrashSummary` object, which combines the information extracted from the crash report with additional data required for the triage.

The resulting `CrashSummary` includes the following elements:

- **Native Stack Trace**: The raw output produced by POIROT, consisting of the sequence of native frames leading to the crash.
- **JNI Bridge Method**: Identified by decoding the JNI signature extracted from the crash, allowing the system to locate the corresponding Java declaration within the APK (Listing 2.2).
- **Java Call Graph**: Obtained by filtering the FlowDroid-generated JCG[1] to retain only the paths relevant to the crash. This helps the LLM analyse the Java-side control flow more easily and increases the coverage of the application during triage.
- **Libraries and methods map**: Using the stack-trace information, the tool identifies which native library contains each frame involved in the crash (including the JNI bridge). This provides the LLM with a compact map of all methods in the stack trace and the `.so` file in which each one resides.

5.3.2 Initialisation and Context Loading

Once the crashes have been normalised, the system prepares the analysis environment by loading the necessary application artefacts into the static-analysis tools. The target APK is opened in Jadx to expose its manifest and Java classes, while the relevant native libraries are imported into Ghidra for disassembly and decompilation.

Note. Although Ghidra supports loading multiple binaries within the same project, doing so significantly increases start-up time and expands the set of libraries that the model may query, often without yielding additional useful information. For this reason, only a filtered subset of libraries is loaded, as described in Section 6.4.1.

At this point no LLM reasoning has been performed yet. This stage is purely preparatory: it ensures that the MCPs can access all artefacts the model may request during the triage process.

5.3.3 Agent Setup

Before any crash is analysed, the LLM agent receives a *system prompt* (shown in Listing A.1), that defines its role as a vulnerability triage agent and constrains its reasoning process. This

[1] `https://github.com/secure-software-engineering/FlowDroid`

prompt outlines the analysis pipeline the model must follow (see Section 5.2), explains how to interpret the inputs provided in the user prompt, and specifies the structure and purpose of the JSON output. It also informs the agent of the available MCPs and clarifies when tool calls should be used.

5.3.4 Crash Prompting

For each crash, a dedicated *user prompt* provides the corresponding `CrashSummary`, which contains all the information required by the LLM to perform the analysis. The `CrashSummary` contains the native stack trace, the filtered JCG leading to the JNI entry point, and the mapping of libraries and methods involved in the fault. This ensures that the model receives a concise yet complete representation of the execution path, covering both the Java and native layers relevant to the crash.

In case of multiple crash of the same method, the agent will remain the same, is only provide with the new crash to classified as a user prompt, but the agent it's not reseted, so it already known some apk/librari stracture. However, a new agent is created whenever the analysis of a new crash begins.

5.3.5 Tool-Mediated Reasoning

During the analysis, the model is free to decide which tool to invoke and when to invoke it, with the objective of retrieving the evidence required for the classification. The system prompt provides a pipeline for collecting information, that serves as a guideline. The agent can adaptively select the most appropriate MCP call based on the evidence needed to advance its reasoning, allowing the triage process to remain flexible while still grounded in verifiable tool-assisted retrieval.

5.3.6 Assessment Generation

Once sufficient evidence about the crash has been gathered, the LLM synthesises a structured assessment, returned as a `VulnResult`. This object follows a strictly defined schema and contains all information required to characterise the crash, justify the classification, and, when applicable, describe a plausible exploitation path.

The `VulnResult` begins with the core elements of the triage:

- **chain_of_thought**: an internal step-by-step reasoning trace produced before the final verdict.
- **is_vulnerable**: the final verdict indicating whether the crash corresponds to a likely vulnerability.
- **confidence**: a score in the range $[0, 1]$ reflecting how strongly the evidence supports the classification.
- **reasons**: a list of short, concrete justifications summarising the logic behind the verdict.
- **cwe_ids**: identifiers of relevant Common Weakness Enumeration, enabling alignment with standard vulnerability taxonomies.
- **severity**: an estimated impact level (low, medium, high, or critical) based on the nature of the fault and the contextual evidence.

To provide a complete picture of the issue, the output also includes:

- **affected_libraries**: the actual native libraries involved in the crash.
- **call_sequence**: the sequence of functions—Java and native—that lead to the faulting code region.
- **evidence**: a collection of structured `EvidenceItem` objects, each containing a function name, address, code excerpt, or decompiled snippet, together with a brief explanatory note.
- **recommendations**: actionable steps or fixes that could mitigate the issue.
- **assumptions** and **limitations**: statements that explicitly document missing information or uncertainty in the reasoning.

If `is_vulnerability = true`, the agent also produces an `Exploit` object describing the exploitability of the issue. Its structure includes:

- **exploitability**: whether exploitation is unknown, theoretical, or practically achievable.
- **trigger_method**: the mechanism required to activate the vulnerability (e.g. malformed input, crafted intent).
- **prerequisites**: environmental or permission-related conditions needed for the exploit to succeed.
- **exploit_pipeline**: an ordered, high-level description of how an attacker could progress from initial conditions to triggering the vulnerable behaviour.
- **PoC_commands**: a theoretical, ready-to-use ADB or shell commands for reproducing the crash or exploit.
- **poc_files**: references to crafted payload files used during exploitation.

The additional components, **Chain-of-Thought (CoT)** and the **Exploit / Chain-of-Process (CoP)** block, were included to guide the LLM toward grounded, non-speculative reasoning.

The **CoT** field stores the model's internal reasoning steps used *before* classification. Its purpose is not to be shown to a human evaluator, but to keep the model focused on systematic, step-by-step analysis rather than relying on pattern-matching shortcuts. By prompting the model to reason explicitly, the system reduces the likelihood of superficial or hallucinated assessments and encourages the analysis of concrete evidence retrieved via MCP tools. Empirically, the inclusion of Chain of Thought (CoT) led to more consistent and accurate classifications, reducing false positives.

The **CoP**/`Exploit` section is designed for a similar reason: it prevents the model from producing arbitrary or unrealistic exploitation scenarios. Instead of guessing, the model must construct a plausible and evidence-driven exploitation flow grounded in the actual structure of the application, the available attack surface, and the retrieved code snippets. Requiring a exploitation pipeline and concrete PoC commands constrains the model to operate within the boundaries of what is technically justified by the evidence.

5.3.7 Final Report Generation

For each crash of application method, the system constructs a `CrashSummary` that is passed to the LLM, enabling a contextualised classification through the Jadx and Ghidra MCPs. The resulting `VulnResult`, together with the corresponding `CrashSummary` and `Statistics` for that crash, is then wrapped into an `AnalysisResult`. A collection of these objects forms an `AnalysisResults`, which represents all crash-level assessments associated with the same method.

Finally, the analysis run is encapsulated in an `AnalysisBlock`, which aggregates:

- the tool-level information (e.g. selected LLM model and tool versions).
- the application `AppMetadata`,
- the complete set of `AnalysisResults` for the application,

5.4 Output Structure

The `AnalysisBlock` is serialised into a well-formed `JSON` file, allowing the entire workflow, from crash extraction to final assessment, to be exported, archived, and automatically processed

by external systems. This enables downstream tasks such as aggregating results across applications, filtering by severity or CWE category, computing statistics, or integrating the outputs into broader testing and validation pipelines.

Each report is written to: `<out-dir>/<pkg>/<JNImethod>/report.json`. Listing B.1 shows an example of a real final JSON output.

The top-level object contains a single field, `analysis`, wrapping all the relevant metadata and results:

- **`analysis.tool`** (`ToolInfo`) Describes the environment and configuration used for the triage:
  - `model_name`: identifier of the LLM used (e.g. `"gpt-5.1"`, `"gpt-oss:120b"`).
  - `apk_path`: path to the analysed APK (e.g. `"APKs/com.tplink.skylight/base.apk"`).
  - `version`: internal version of the triage tool.
- **`analysis.app`** (`AppMetadata`) Contains the application metadata extracted from Jadx:
  - `app_name`: human-readable application label.
  - `package`: package name (application identifier).
  - `min_sdk`, `target_sdk`: minimum and target Android SDK levels.
  - `version_name`, `version_code`: versioning information of the analysed build.
- **`analysis.analysisResults`** (`AnalysisResults`) A *list* of per-crash assessments. Each element is an `AnalysisResult` object bundling:
  - `crash` (`CrashSummary`): a normalised description of the crash:
    - `ProcessTermination`: crash cause as reported by the runtime..
    - `StackTrace`: list of native frames involved in the crash.
    - `JavaCallGraph`: Java → JNI call chain leading to the JNI bridge method.
    - `JNIBridgeMethod`: JNI entry point associated with the crash.
    - `JavaCallGraph`: Java call chain leading to the JNI method.
    - `FuzzHarnessEntry`: fuzzer entry function used to drive inputs.
    - `ProgramEntry`: process entry point.
    - `LibMap`: native libraries involved in the crash.
  - `assessment` (`VulnResult`): the vulnerability triage produced by the LLM:
    - `chain_of_thought`: a list of strings representing the step-by-step monologue that LLM thinks through before classifying.

    * `is_vulnerable`: boolean verdict indicating whether the crash is likely a real vulnerability.
    * `confidence`: numerical confidence in $[0, 1]$ associated with the verdict.
    * `reasons`: short textual bullets explaining the decision (e.g. missing null-termination, out-of-bounds read).
    * `cwe_ids`: list of relevant CWE identifiers (e.g. `"CWE-125"` for out-of-bounds read).
    * `severity`: estimated impact level (`"low"`, `"medium"`, `"high"`, `"critical"` or `null` if unknown).
    * `affected_libraries`: list of libraries implicated in the crash.
    * `recommendations`: concrete mitigation or follow-up actions (e.g. enforcing null-termination, adding bounds checks).
    * `assumptions`: explicit assumptions made by the model (e.g. nature of the input or control over certain parameters).
    * `limitations`: known gaps in the analysis (e.g. partial decompilation, missing source, lack of full context).
    * `evidence`: list of `EvidenceItem` objects, each optionally containing:
        - `function`: function name relevant to the issue.
        - `address`: code address within the library (when available).
        - `file`: library or source file associated with the evidence.
        - `snippet`: short decompiled excerpt or code fragment.
        - `note`: explanation of why this snippet supports the classification

- `Statistics`: basic metrics on the analysis.
    * `time`: total analysis time.
    * `llm_requests`: number of LLM requests.
    * `llm_tool_calls`: number of MCP tool calls.
    * `input_tokens`: tokens sent to the LLM.
    * `output_tokens`: tokens produced by the LLM.

6 Implementation

The implementation realises the design described in Chapter 5 through a Python-based orchestration layer built around three core elements: (i) structured data models, (ii) helper functions for integrating the MCP, and (iii) specialised prompting templates to ensure reproducible and deterministic behaviour across crash analyses. It also describes the concrete system architecture, library dependencies, internal module structure, and the integration workflow between the LLM, Jadx/Ghidra MCPs, and POIROT's crash artefacts.

The full source code is available at `github.com/Nicola-01/LLM-Triaging`.

6.1 Libraries and Dependencies

Main libraries and tools adopted in this thesis:

- **pydantic (v2.12.3)** Used for strict, typed data modelling. Every component in the triage pipeline, from crash summaries to tool responses and vulnerability assessments, conforms to well-defined Pydantic classes. This guarantees deterministic parsing of the LLM outputs, ensures structural validation of all intermediate artefacts, and allows lossless serialisation and deserialisation into JSON.
- **pydantic_ai** Provides the agent abstraction, the MCP client and the `MCPServerStdio` transport, enabling the program to communicate with Jadx and Ghidra MCPs over standard I/O streams.

- **Jadx** and **Ghidra** The reverse-engineering backends operate as standalone MCPs server. Jadx exposes bytecode, manifest, and resource-level information. Ghidra exposes disassembly, function listings, and decompilation output.

The system also depends on the POIROT output directory produced during fuzzing.

6.2 Program Structure

The codebase is structured around clear modular boundaries, as illustrated in Figure 6.1.

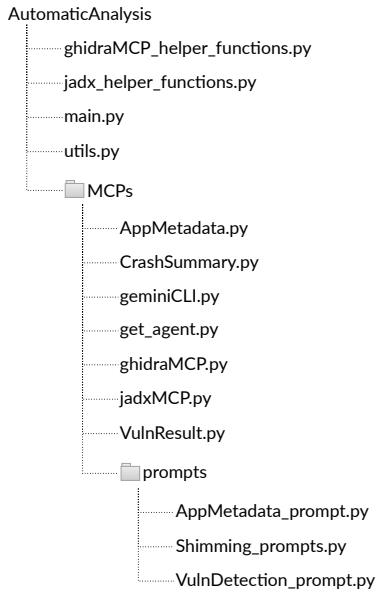


Figure 6.1: Structure of the implementation, modules, data flow, and orchestration logic.

Entry Point and Orchestration

The main program (`main.py`) performs the following tasks:

1. parses CLI arguments for model configuration, timeout, and output path;
2. For each apk:
   (a) loads POIROT crash artefacts;
   (b) instantiates the LLM triage agent through `MCPs/get_agent.py`;
   (c) coordinates the full pipeline:
   load app crashes → load APK / library context → LLM triage → report
   (d) generates the structured JSON report.

In the same directory as the main program, several utility and helper modules are provided. The Ghidra and Jadx helper functions manage the corresponding GUI sessions and import of APKs and native libraries into their analysis environments.

MCP Helper Functions and Data Structures

The `MCPs/` directory contains both the core data-structure definitions used throughout the triage pipeline and a set of helper functions that support MCP-based tool interaction. The data models defined in this module, such as `CrashSummary`, `AppMetadata`, `VulnResult`, `Exploit`, and `AnalysisBlock`, are those described in Chapter 5.

The directory also includes helper utilities that streamline communication with the Jadx and Ghidra MCPs. These modules handle the additional initialisation steps required to make the MCPs available to the LLM, ensuring that each backend is correctly configured before the analysis begins.

Prompt Specialisation

The directory `MCPs/prompts/` contains the family of specialised prompt templates used by the triage agent. Each prompt serves a distinct function and is tailored to the type of information the LLM must extract or analyse.

- `AppMetadata_prompt.py` Defines the extraction rules and acceptable formats for retrieving application metadata from Jadx. The corresponding system prompt is provided in Listing 6.1.

- `VulnDetection_prompt.py` Encodes the complete vulnerability-triage logic. It specifies what constitutes a vulnerability, the backward data-flow reasoning process, the conditions for exploitability, and the full JSON schema required for the output.
  A variant of this prompt is used when the JCG is unavailable; in this case, the model is not required to reason about the Java-side control flow. Listing A.1 shows the full prompt.

- `Shimming_prompts.py` Defines the system prompts used when interacting with models that lack native MCP support. The template includes placeholders into which the required tool descriptions and system instructions are dynamically injected. It also enforces the appropriate output schema, either metadata extraction or vulnerability assess-

ment, ensuring consistent and predictable behaviour across different backends. Further details are provided in Section 6.3.2.

```
You are a Jadx MCP assistant. Your goal is to extract app metadata from the
APK currently open in Jadx.

Steps (use MCP tools where applicable):
1) Get the AndroidManifest.xml (tool: `get_android_manifest`).
2) Extract the package name from <manifest package="...">.
3) Extract application label:
   - If <application android:label="..."> is a literal, use it.
   - prefer using `search_string` then `get_strings`.
4) Extract SDK info from <uses-sdk> if present (minSdkVersion,
    targetSdkVersion).
5) If versionName or versionCode are available (manifest or packageInfo),
    include them.

Respond ONLY by populating the output schema.
```

Listing 6.1: System prompt used for extracting application metadata via Jadx MCP.

6.3 LLM and MCP Integration

The system is LLM-agnostic: it is not tied to a single model and supports multiple alternatives. This flexibility allows users to select the model that best fits the task, without depending on any specific provider. Ensuring that the same triage logic can operate with different LLM vendors or even fully local models. Currently, the program supports `OpenAI`, `Gemini` (and Gemini-cli), and local `Ollama` models, and can be extended to include additional vendors.

The integration between the LLM and the reverse-engineering backends is realised via the `pydantic_ai` agent framework and the standardised MCP interface. This section describes the mechanisms used to initialise the agent, connect to the MCPs, and enforce protocol correctness.

6.3.1 Agent Setup

The agent is created through the function `get_agent` defined in `MCPs/get_agent.py`. This function instantiates a `pydantic_ai.Agent` object that coordinates the chosen LLM and a list of MCP toolsets (`MCPServerStdio`). The agent receives:

- *A system prompt*, which defines its high-level role (e.g. vulnerability assessor or metadata extractor) and encodes the behavioural constraints that the model must follow. This includes the JSON-only policy, the allowed response formats, and the list of tool names that the LLM may invoke.

- *An output type*, such as `AppMetadata` or `VulnResult`. This schema determines how the final answer must be structured and enables automatic validation of the model's output.

- A list of connected *MCP servers*, each represented by a `MCPServerStdio` instance. These servers expose the reverse-engineering capabilities that the agent can access whenever the model chooses to perform a tool call.

The function automatically selects the appropriate model provider according to the prefix of the model name supplied at start-up:

- names beginning with `gpt-` initialise an `OpenAIChatModel` via the `OpenAIProvider`;

- names beginning with `gemini-` initialise a `GoogleModel` via the `GoogleProvider`;

- any other identifier defaults to a locally hosted model served through the `OllamaProvider`.

This logic allows seamless switching between cloud-based and local models.

The agent also keeps track of the number of tool calls executed during the analysis, as well as the token usage associated with each step, both the input tokens sent to the model and the output tokens generated in response, enabling detailed monitoring of the interaction.

Some models require a different integration setup.

Gemini-CLI. If the model name is specified as `gemini-cli`, the program interacts with the Gemini Command-Line Interface[1] instead of the standard API endpoint.

[1] https://geminicli.com/

The integration is handled by the helper module `MCPs/geminiCLI.py`, which provides two key functions: `query_gemini_cli()` and `gemini_response_parser()`. The function `query_gemini_cli()` builds a unified text prompt combining the system and user contexts, executes the `gemini` command via subprocess. The model's response is then passed to `gemini_response_parser()`, which cleans the raw CLI output and attempts to parse the result as JSON. If the output cannot be validated against the expected schema, the function retries; after four failed attempts, an error is raised.

Local LLM Backends Local LLMs, particularly open-source models run through systems such as Ollama, do not provide native support for the MCP protocol. When used directly, these models frequently violate the protocol by hallucinating tool calls, mixing JSON with free text, or returning incomplete or malformed outputs. To ensure protocol-correct behaviour, a dedicated *shimming layer* was introduced.

The agent therefore instantiates an `oss_model` backed by `MCPs/shimming_agent.py`, which acts as a controlled execution environment. Instead of allowing the model to interact freely with Jadx and Ghidra, the shimming agent constrains all communication to a strict JSON structure and enforces disciplined tool use. A detailed description of this mechanism is provided in Chapter 6.3.2.

6.3.2 MCP setup

The use of MCPs requires additional setup steps, and the exact procedure depends on the execution environment in which the LLM operates. Different backends, cloud-based services, the command-line `gemini-cli` interface, or local offline engines such as Ollama, require different integration strategies. As a result, the orchestration layer must initialise and manage the MCP servers in a manner that is compatible with the chosen execution backend, ensuring that the LLM can reliably invoke tool operations throughout the triage process.

Jadx

The integration of Jadx into the pipeline relies on the `jadx-ai-mcp`[2] server, which exposes Jadx functionalities.

Unlike Ghidra, Jadx does not require any additional configuration steps once the MCP plugin is installed.

[2] `https://github.com/zinja-coder/jadx-ai-mcp`

The server becomes available as soon as the Jadx GUI instance is running, and de designated port is free. Therefore, when analysis of an APK begins, the program launches a dedicated Jadx session using the command: `jadx-gui <apk>`. This opens the GUI instance and allows the `jadx-ai-mcp` plugin to start its server.

Session management is handled by an helper module `jadx_helper_functions.py`, which contains utilities to start, monitor, and terminate the Jadx process.
The function `start_jadx_gui()` launches the GUI, monitors its output stream, and blocks execution until the MCP server is detected or a timeout occurs. If Jadx fails to start within the configured timeout, the function terminates the process and aborts the analysis. The module provides `kill_jadx()`, used to gracefully stop the Jadx process when it is no longer required or when switching to a different APK.

Cloud-based backends. For cloud-based LLM backends (e.g. GPT-5.1 or Gemini-3), integration with MCP servers can rely directly on the Pydantic AI framework[3]. Pydantic AI provides the `MCPServerStdio` utility, which allows the orchestration layer to launch and manage MCP servers as subprocesses, including the startup command and initialisation timeout.

Listing 6.2 shows the initialisation of the Jadx MCP server, where an `MCPServerStdio` instance is created to launch `jadx_mcp_server.py` from the directory specified by the system variable `JADX_MCP_DIR`.

Once the `MCPServerStdio` is running, the agent connects to the `jadx-ai-mcp` endpoint to query project-level information. Listing 6.3 shows how the system retrieves the APK metadata using a dedicated agent call.

```
1 def make_jadx_server(timeout: int = 60) -> MCPServerStdio:
2     return MCPServerStdio(
3         "uv",
4         args=["--directory", os.getenv("JADX_MCP_DIR"), "run", "
    jadx_mcp_server.py"],
5         timeout=timeout,
6     )
```

Listing 6.2: Function `make_jadx_server` used to initialise the Jadx MCP connection

[3] https://docs.pydantic.dev/latest/

```
server: MCPServerStdio = make_jadx_server()
...
async with get_agent(JADX_APP_METADATA, AppMetadata, [server], model_name=
    model_name) as j_agent:
    j_meta = await j_agent.run("Extract app metadata from the currently open
    Jadx project.")
appMetadata: AppMetadata = j_meta.output
...
```

Listing 6.3: Method `get_jadx_metadata` that returns the app metadata from Jadx

`jadx-ai-mcp` TOOLS. The Jadx MCP server provides several groups of tools that allow the agent to inspect and navigate the Java and Android components of the application:

- **Manifest and Entry-Point Retrieval:** Extract information from the Android manifest and identify key components such as activities, application classes, and initial entry points.
- **Class and Method Inspection:** Explore the full set of classes, access source code, and inspect methods, enabling structural analysis of the Java layer.
- **Search and Low-Level Views:** Locate methods by name—including JNI bridge functions—and retrieve Smali representations for low-level inspection.
- **Resource Access:** Retrieve resources, configuration files, and embedded strings, useful for understanding app behaviour or identifying protocol- or input-related logic.
- **GUI-Assisted Context Capture:** Interact with elements currently selected in the Jadx GUI, allowing the agent to access context-sensitive views during analysis.
- **Refactoring Utilities:** Optional renaming tools that improve readability within Jadx without affecting the underlying binary or the evidence used for the final report.

GHIDRA

The integration of Ghidra into the pipeline relies on the `GhidraMCP`[4], which exposes Ghidra's analysis and decompilation functionalities.

[4] `https://github.com/LaurieWired/GhidraMCP`

Unlike Jadx, Ghidra cannot automatically create projects or import binaries directly from the command line, which makes its initial setup less straightforward. To address this limitation, the pipeline employs `ghidra-cli`[5], a wrapper script that allows the creation of a new project and the import of native libraries using a simple command such as: `ghidra-cli -n -i file.so` This command launches the Ghidra GUI (see Figure 6.2) and automatically imports the specified binaries.

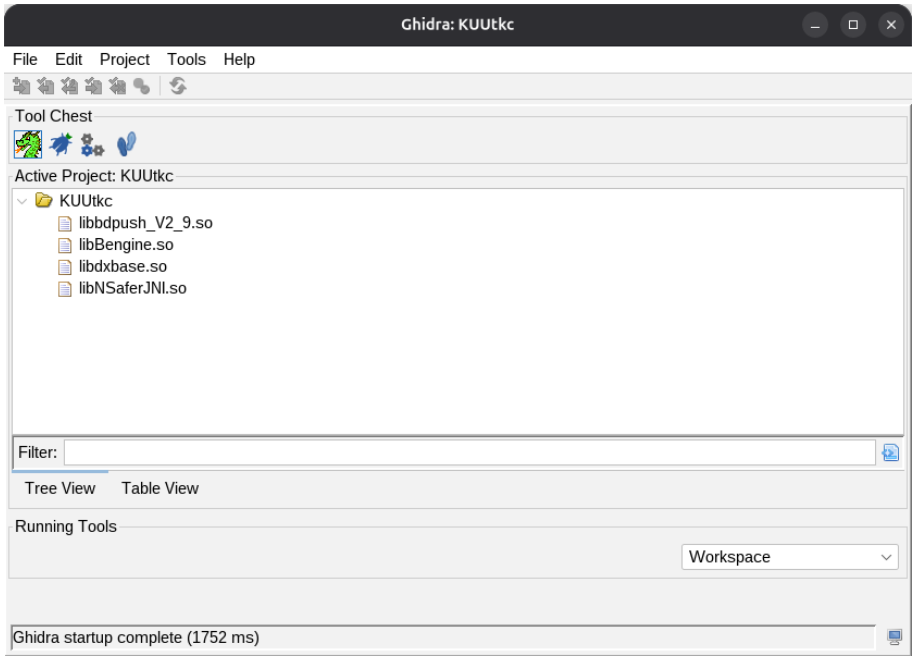


Figure 6.2: Ghidra GUI after being launched via `ghidra-cli`, with multiple .so files.

However, the Ghidra MCP server only becomes available once an imported binary is opened inside the CodeBrowser (interface opened by the dragon icon), and this action cannot be performed via the command line.

To fully automate this process, the helper module `ghidraMCP_helper_functions.py` was developed. This module uses `wmctrl` to manage desktop windows and `pyautogui` to simulate keyboard interactions (e.g. `Tab`, `Down Arrow`, `Enter`) in order to programmatically select and open imported binaries within the GUI. It also supports opening the Ghidra GUI with multiple imported files, switching between them programmatically, and closing the currently open file before loading a new one.

[5] `https://github.com/Denloob/ghidra-cli`

Since the MCP server is bound to the active CodeBrowser instance, the LLM can only retrieve information from a single `.so` file at any given time. Running multiple CodeBrowser instances is not possible because they would share the same MCP port. However, real applications often rely on multiple native libraries, and analysing them sequentially requires explicit control over which binary is currently active in Ghidra.

To address this limitation, the base `GhidraMCP` implementation was extended with three additional methods:

- `list_available_libs()`: Returns the list of all native libraries currently imported into the Ghidra project.
- `get_current_lib_name()`: Identifies the library that is currently open in the Ghidra CodeBrowser and therefore exposed through the MCP server.
- `open_lib(lib_name: str)`: Switches the active view in the CodeBrowser to the specified library. This operation brings the selected binary into focus so that subsequent MCP calls operate on the intended file.
- `open_external_method(method_name: str)`: Allows the agent to analyse methods that reside in libraries not currently loaded into Ghidra. When invoked, this function closes the active Ghidra GUI instance, identifies the native library that defines the requested method, and reopens Ghidra with that library included among the imported files. This ensures that the LLM can retrieve code and metadata for methods located outside the initially loaded set of libraries, without requiring manual intervention.

These extensions enable controlled, sequential analysis of multiple native libraries while maintaining compatibility with Ghidra's single-CodeBrowser MCP constraint.

Once the Ghidra GUI environment is ready, the triage agent establishes communication with the Ghidra MCP server through the `make_ghidra_server()` function defined in `ghidraMCP.py`.

`GhidraMCP` TOOLS. The Ghidra MCP server exposes several groups of analysis capabilities that the triage agent can invoke during native-code inspection:

- **Decompilation and Disassembly:** Provide low-level assembly views and high-level decompiled C-like code, enabling the agent to inspect function bodies and reconstruct program behaviour.

- **Function and Address Queries:** Allow the agent to retrieve functions, addresses, and cross-references, making it possible to trace control flow and understand how different parts of the binary interact.

- **Listing and Enumeration:** Return structured listings of functions, classes, symbols, segments, strings, and other program components, giving the agent a global overview of the binary.

- **Search and Cross-References:** Support targeted lookups (e.g., by name or substring) and navigation of cross-references, helping the agent locate relevant code regions linked to the crash.

- **Editing and Annotation:** Enable renaming, prototype adjustments, and commenting within the decompiler or disassembly views, improving clarity during iterative analysis.

MCP on Gemini-cli and Open Source LLM

When the selected model is neither a cloud-based GPT nor Gemini model, the system falls back to a local execution strategy. Two pathways are implemented: one based on `gemini-cli` and one based on open-source models served through `Ollama`. Both approaches differ from the standard `MCPServerStdio`-based configuration.

Gemini-cli. The integration of Gemini models is handled through the standalone `gemini` command-line interface. Instead of establishing a persistent MCP transport via `MCPServerStdio`, the system invokes `gemini-cli` directly; it is then `gemini-cli` itself that, based on its configuration file `.gemini/settings.json`, automatically launches the required MCP servers. When `gemini-cli` starts, it spawns the corresponding MCP processes, after which the triage agent can immediately access the exposed tool endpoints.

Open-source models via Ollama. Most open-source LLMs only understand the abstract concept of "tool calls", typically as JSON structured actions, but they do not enforce the strict turn-based protocol and message discipline required by the MCP. For this reason, the pipeline incorporates a dedicated *shimming layer* implemented in `shimming_agent.py`. This mechanism transparently enables the use of MCP tools with models that lack MCP support.

Shimming layer. The shimming agent wraps the open-source model in a supervised execution loop that enforces deterministic, protocol-correct behaviour. Its role is to ensure that models without native MCP support can still interact reliably with external tools by constraining their outputs to a strict, verifiable format.

The system prompt of the shimming agent specifies that every model response must be a single JSON object containing either:

- a tool invocation, e.g. `{"action": "<tool_name>", "args": {...}}`, or
- a final answer, e.g. `{"action": "final", "result": {...}}`.

When the model outputs an action, the Python wrapper executes the corresponding tool through the `BasicMCPClient` provided by the `llama_index` MCP utilities, forwards the result back to the model, and continues the interaction loop.

Only when the model emits `"action": "final"` does the agent proceed to validating the output against the expected schema (e.g. `AppMetadata`, `VulnResult`).

During a shimming-mediated interaction, the execution loop proceeds as follows:

1. the model receives the available tools through the system prompt;
2. then proposes a tool call in JSON form;
3. the shimmer forwards the request to the appropriate MCP server (Jadx or Ghidra);
4. the server returns structured evidence;
5. the shimmer returns this evidence to the model as the next input;
6. the model decides whether additional evidence is required or whether it can produce the final structured output.

If the final JSON object is not well-formed or does not match the target schema, the model is prompted to generate a corrected response instead of terminating the execution.

The purpose of the shimming layer is to mediate the interaction in a way that prevents protocol violations and ensures that local open-source models behave like fully MCP-compatible agents.

The following pseudo-prompt and pseudo-code summarises the control flow implemented by the shimming layer.

```
1 You are a tool-using assistant that can use tools to ...
2 You can ONLY communicate in JSON.
3
4 Available functionalities:
5 < Schema of MCP's tools >
6
```

```
For each step, reply ONLY with a valid JSON with the proposed schema.
{"action": <tool_name>,  "args": { ... }}

After receiving the tool results, you will be asked again.
Only when explicitly instructed with "final" may you return your writeup:
{"action": "final", "result": <writeup>}

Rules:
- NEVER output text outside of JSON.
- NEVER skip directly to "final" without using a tool.
```

Listing 6.4: Pseudo-prompt of the shimming layer, label=lst:pseudoPrompt

```
procedure ShimmingAgent(user):
    initialise model with system prompt
    initialise empty dialogue history

    loop:
        append user prompt to history
        model_output ← query model(history)

        json ← validate_and_extract_JSON(model_output)
        if json is invalid:
            prompt ← "Invalid JSON. Try again."
            continue

        if json.action == "final":
            return json.result

        result ← execute_tool(json.action, json.args)

        if result is empty:
            prompt ← "Malformed tool call. Try another."
        else:
            prompt ← "Tool Response: " + serialize(result)
```

Listing 6.5: High-level pseudo-code of the shimming layer

6.4 POIROT Output & Processing

As mentioned in Chapter 4, the triage procedure begins from the output generated by POIROT. During its *Triage phase*[6], POIROT produces a file named `folder2backtraces.txt`, shown in Listing 4.1. This file serves as the primary input for the analysis performed in this project. After running POIROT on a batch of applications, the resulting directory structure for the target APK appears as illustrated in Figure 6.3.

Figure 6.3: Directory structure of the target APKs as produced by POIROT, including extracted libraries and reproduced crash outputs.

For each package contained in `target_APK`, the program explores the `fuzzing_output` directory and identifies all entries named `<functionName-Signature>` that include the file `folder2backtraces.txt`. From this structure, the program builds a map in which the key is the package name (`pkgName`) and the value is a list of backtrace files. This mapping is then used to perform the vulnerability triage.

6.4.1 Extraction of Native Libraries

POIROT already extracts the native libraries as part of its output. As shown in Figure 6.3, the `lib/` directory contains the compiled native binaries, organised by Application Binary Interfaces (ABIs) (e.g. `arm64-v8a`, `x86_64`). The triage pipeline selects which variant to analyse using a predefined ABI preference list; in most cases, `arm64-v8a` is prioritised.

For each extracted library, the program enumerates the available symbols using the `nm` utility, which "displays information about symbols in the specified file, which can be an object file, an executable file, or an object-file library" [56]. The resulting symbol table is cross-referenced

[6] `https://github.com/HexHive/droidot/#triage`

with the crash stack trace in order to identify only those libraries that appear in the execution path. This filtering step avoids unnecessary MCP queries and reduces the amount of contextual information that must be supplied to the LLM. Moreover, Ghidra cannot reliably open a large number of libraries within the same project, making it essential to limit the analysis to the subset of binaries that are actually relevant to the crash.

If additional libraries become relevant during the analysis, the agent can still access them on demand through the `open_external_method` MCP function. This mechanism allows the LLM to load previously excluded native binaries autonomously, ensuring that methods outside the initial import set can be inspected without manual intervention.

The resulting filtered map is then recorded in the `CrashSummary` object associated with the crash. This enriched crash descriptor is subsequently passed to the triage agent, ensuring that the analysis stage focuses exclusively on the libraries that may have contributed to the fault.

6.4.2 Java Call Graph

A second input provided to the model through the `CrashSummary` object is the JCG. The purpose of this artefact is to reconstruct the sequence of Java method invocations that lead to the JNI entry point responsible for calling the native function in which the crash occurred. While the native stack trace already describes the execution path within the `.so` file, the JCG complements it with the higher-level control flow within the application.

The extraction is performed in two phases:

1. **Generation of a global flow graph.** A complete flow graph of the application is first generated using FlowDroid[7], which statically analyses the APK and produces a comprehensive JCG covering all possible execution paths. The resulting structure is stored in a large JSON file.
2. **Construction of a filtered call graph.** Starting from the complete FlowDroid output, only the call sequence that reaches the JNI bridge associated with the crash are kept. The result is a compact filtered JCG containing the precise Java-level call chain leading to the native entry point.

This filtered JCG is then attached to the `CrashSummary`, enabling the LLM to reason about both native- and Java-level execution paths. In this way, the model can reconstruct the complete control-flow chain leading to the fault, rather than relying solely on the native stack trace generated by POIROT.

[7] https://github.com/secure-software-engineering/FlowDroid

7 Evaluation

This chapter evaluates the effectiveness, reliability, and overall behaviour of the proposed triage system. The goal is to assess how well the LLM-based approach classifies crashes and how the availability of additional program context, i.e. JCG, influences its decisions. The chapter presents the experimental setup, the selected test applications, and a quantitative evaluation based on standard classification metrics. The results provide insight into the strengths and limitations of the current implementation and form the basis for the discussion in the following chapter.

7.1 Experimental Setup

The evaluation of the tool was carried out on a local machine. The experimental environment consisted of:

- Local machine running **Ubuntu 25.04**;
- **Ghidra 11.4.2** (latest version at the time of evaluation);
- **Jadx 1.5.3** (latest version at the time of evaluation);
- **GhidraMCP 1.4** (latest version at the time of evaluation);
- **jadx-ai-mcp 4.0.0** (latest version at the time of evaluation).

7.1.1 AWS for POIROT

The extraction of crashes used for this evaluation is described in Chapter 4, and the corresponding execution environment is detailed in Chapter 4.2. All POIROT runs were executed on an AWS EC2 instance configured for large-scale fuzzing.

7.1.2 LLM Used

The evaluation was conducted using the OpenAI model GPT-5.1. This model represented one of the most capable LLMs available at the time of testing and offered reliable support for MCP-based tool use. GPT-5.1 also integrates smoothly with the Pydantic framework, allowing strict control over structured outputs.

7.2 Test Applications

A total of 137 method were used for the evaluation, originating from 80 distinct applications. Among these 137 cases, 62 crashes involved native methods that were reachable from the Java layer and could therefore be analysed using the filtered JCG.
The remaining 75 crashes involved methods that were either not reachable from Java or for which FlowDroid failed to generate a valid call graph. These cases were still processed by the triage pipeline, but without Java-side context.

Table 7.1 summarises all applications used in the evaluation and reports, for each APK, the metrics collected during the triage process. The columns are defined as follows:

- **App** and **Version**: package name and exact APK version analysed.
- **Methods**: total number of native methods evaluated for that application.
- **Crashes**: number of distinct crashes produced by POIROT; a single method may generate multiple crashes.
- **Time (MM:SS)**: total LLM analysis time for the application (excluding Ghidra startup time).
- **Input token** and **Output token**: total token consumption across all LLM requests for that application.
- **LLM request**: number of request that the model sent to the API, during the full triage stage.
- **LLM Tool Calls**: number of successful MCP tool invocations (Jadx or Ghidra) performed during the analysis.

On average, the analysis time per application was 2 minutes and 38 seconds. Normalising by method rather than by application shows that each method produced an average of 2.13 crashes, with a mean processing time of 42 seconds per method. This indicates that the system is able to process individual crash reports with very low latency, especially when compared to manual inspection workflows. This make the pipeline suitable for large-scale, automated vulnerability triage.

App	Version	Methods	Crashes	Time (M:S)	Input Token	Output Token	LLM request	LLM Tool Calls
br.com.pedidos10	1.16.4	1	2	01:20	36344	4006	5	9
br.com.sulamerica.sam.saude	7.66.0	1	2	01:33	59737	4611	6	16
ca.radioplayer.android	6.3.420.1	1	10	08:00	494465	23484	54	73
chk.kingnet.app	2.10.0	1	2	01:33	57633	5128	9	10
com.ahnlab.v3mobileplus	2.5.20.10	4	7	06:13	415345	20700	34	51
com.amazon.avod.thirdpartyclient	3.0.343.77747	1	4	02:07	155907	8867	16	12
com.android.chrome	115.0.5790.166	2	2	00:44	47771	2293	8	4
com.appgeneration.itunerfree	9.3.13	1	2	01:53	53776	6516	6	14
com.bitstrips.imoji	11.79.0.9763	1	1	00:33	12157	1274	2	1
com.btckorea.bithumb	3.0.2	1	1	00:26	13001	2572	2	3
com.cisco.webex.meetings	45.3.0	1	3	02:09	92548	7643	10	23
com.clearchannel.iheartradio.controller	10.36.0	1	3	03:20	151439	7039	17	20
com.cyworld.camera	4.4.1	2	4	02:07	64943	5945	7	12
com.didiglobal.driver	7.5.88	1	8	06:48	215355	14366	31	41
com.elevenst	9.3.4	1	1	01:14	358406	2707	9	9
com.ford.fordpass	4.23.1	1	4	03:34	198312	10642	17	42
com.ford.fordpasseu	4.23.1	1	3	03:05	84360	6811	9	26
com.google.android.apps.translate	7.7.0.540337148.2	1	1	00:43	21799	1769	3	2
com.hyundaicard.cultureapp	1.0.72	1	1	00:48	14153	2152	2	3
com.intsig.BCRLite	7.85.5.20251016	6	9	05:41	195757	18587	29	56
com.jeju.genie	2.2.13	1	1	00:33	12987	2244	2	3
com.kakaopay.app	2.5.4	1	1	00:28	22759	1863	3	2
com.kartatech.karta.gps	2.44.02	1	1	00:37	31715	1993	4	8
com.kbankwith.smartbank	1.5.8	10	39	24:36	1185064	105964	145	253
com.kbstar.kbbridge	1.2.1	1	1	00:35	13014	2189	2	3
com.kii.safe	12.2.0	3	5	02:50	121176	8035	16	19
com.kt.ktauth	02.01.37	3	7	05:03	178725	15975	27	37
com.mapfactor.navigator	7.3.17	2	3	01:02	59737	4296	9	12
com.microsoft.office.outlook	4.2330.0	1	1	01:34	19565	2182	3	5
com.mitake.android.bk.tcb	3.21.0105	1	2	01:03	49766	3710	8	10
com.mttnow.droid.easyjet	2.70.0	2	3	01:27	80463	5835	11	9
com.pandora.android	2509.1	1	4	07:00	70891	9638	8	20
com.qustodio.parental.control.app.screentime	182.16.0	1	2	01:20	44536	5596	6	11
com.realmestore.app	1.9.0	1	1	00:43	27834	1979	4	10
com.riffsy.FBMGIFApp	2.1.61	1	1	00:32	34696	2000	5	5
com.rockbite.deeptown	6.2.10	1	1	00:17	6564	1421	1	2
com.rstgames.durak	1.9.11	1	1	00:36	19232	1726	3	3
com.samsungcard.shopping	1.4.901	1	1	00:22	12380	1170	2	1
com.shareitagain.whatslov.app	12.5.0	1	4	02:04	161927	7541	17	24
com.skmc.okcashbag.home_google	7.0.8	3	4	02:56	93879	10606	13	22

App	Version	Methods	Crashes	Time (M:S)	Input Token	Output Token	LLM request	LLM Tool Calls
com.skt.prod.dialer	13.6.5	2	7	04:38	212223	12100	26	26
com.skt.smartbill	6.4.1	5	16	09:21	378136	28641	54	77
com.skysoft.kkbox.android	6.4.60	2	3	02:21	74898	8196	11	12
com.smg.spbs	3.42	1	1	00:38	19257	2338	3	6
com.sony.tvsideview.phone	6.2.0	1	1	00:57	56892	2387	4	10
com.sopheos.videgreniersmobile	20.11.10	1	2	01:00	48392	3213	6	10
com.ss.android.ugc.trill	9.1.5	2	2	00:58	52594	3785	8	6
com.ssg.serviceapp.android.egiftcertificate	2.6.10	1	1	00:31	18401	1734	3	2
com.teamjin.deliveryk	7.0.3	1	2	01:23	31816	4143	4	9
com.telkomsel.roli	3.1.0	2	2	01:58	107680	4364	8	8
com.tencent.mm	8.0.28	9	41	28:21	2752831	94404	204	239
com.tmon	5.8.4	2	3	02:19	85577	5551	11	15
com.tplink.skylight	3.1.20	1	2	01:09	59677	5617	7	14
com.ucturbo	1.10.3.900	1	1	00:31	22220	1869	3	2
com.youdao.hindict	6.6.2	1	2	00:47	38715	2880	6	4
fr.radioplayer.android	6.6.420.1	1	9	10:28	202292	21668	22	59
heartratemonitor.heartrate.pulse.pulseapp	1.2.7	1	1	00:34	25401	2109	3	2
kr.co.busanbank.mbp	3.0.10	6	19	15:20	520673	44981	59	112
kr.co.kfcc.mobilebank	1.2.6	1	1	00:30	13046	1827	2	2
kr.co.morpheus.geps	02.42	1	1	00:59	22248	1544	3	4
kr.co.samsungcard.mpocket	5.4.306	2	2	00:55	29228	2915	4	8
kr.go.iros	1.2.3	1	1	00:43	24988	2374	4	4
kr.go.kcs.mobile.pubservice	1.0.281	2	2	01:00	25362	3232	4	4
kr.go.minwon.m	2.5.95	1	1	00:30	14516	1378	2	4
kr.go.nts.android	12.8	2	3	01:27	59025	4887	8	13
kr.go.wetax.android	5.4.14	2	2	01:30	27318	4586	4	5
net.ib.android.smcard	10.1.903	2	2	01:04	51170	4139	8	7
net.orizinal.subway	3.7.8	4	4	02:33	434088	9043	23	19
nh.smart.allonebank	1.8.8	1	1	00:36	24960	1695	4	4
nh.smart.banking	4.1.2	3	5	02:58	108685	10359	17	18
nh.smart.card	6.5.0	1	1	00:30	14111	1783	2	3
nh.smart.nhallonepay	3.3.7	2	2	01:25	84244	4386	12	11
nh.smart.nhcok	2.0.70	1	1	00:37	14361	1676	2	4
ragazzo.alphacode.com.br	3.0.9	1	2	01:27	38671	3792	5	13
ru.cn.tv	7.8.16	1	1	00:20	20412	1492	3	4
ru.mw	4.50.0	1	1	00:36	24321	2164	4	7
sam.myanycar.samsungFire	6.0.8	1	1	00:40	20620	2394	3	5
stickermaker.whatsappstickers	1.01.40.02.08	3	3	02:16	78127	7106	12	17
tw.com.taishinbank.ccapp	5.631	1	2	01:38	31781	4205	4	10
tw.gov.tra.twtraffic	2.1.2	1	2	01:18	42834	3530	5	13
Averaged values across all applications:		**02:38**	**1.76**	**3.76**	**133581.92**	**8364.95**	**13.89**	**20.44**

Table 7.1: Summary of applications used in the evaluation, including per-app method counts, crash counts, processing time, token usage, LLM and tool invocations.

7.3 Evaluation Metrics

To objectively assess the performance of the LLM-based triage system, standard information retrieval metrics derived from a confusion matrix are employed. Given the binary nature of the classification task, determining whether a crash is a vulnerability or not, the possible outcomes are defined as follows:

- **True Positive (TP):** A vulnerability correctly identified by the model. The crash represents a security risk, and the system correctly classified it as such (`is_vulnerable: true`).
- **False Positive (FP):** A benign crash incorrectly flagged as a vulnerability. This represents a "false alarm", where the model assigns security relevance to a non-exploitable issue.
- **True Negative (TN):** A benign crash correctly identified as non-vulnerable. The system correctly determined that the issue does not pose a security risk.
- **False Negative (FN):** A vulnerability missed by the model. The crash is dangerous, but the system incorrectly classified it as benign.

Based on these definitions, the following metrics were calculated:

Accuracy Accuracy quantifies how many of the analysed crashes were assigned the correct label. It provides an overall view of the system's correctness across both vulnerable and non-vulnerable cases.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

Precision Precision quantifies the reliability of the positive predictions. Indicates how often a crash flagged as vulnerable is indeed a real vulnerability.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Recall Recall measures how effectively the system identifies all crashes that correspond to real vulnerabilities. Captures the model's ability to avoid missing true security-relevant issues. Indicates how many actual vulnerabilities the system successfully detects.

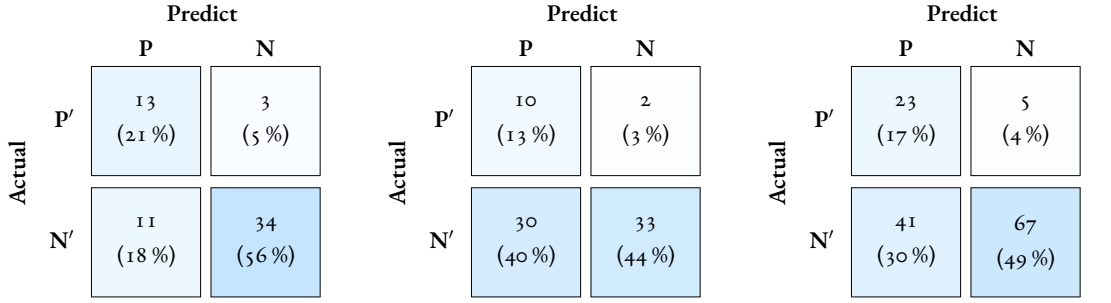
$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

F1-Score The F1-score provides a single value that captures the trade-off between precision and recall. It reflects the balance between detecting vulnerabilities and avoiding false alarms.

$$\text{F1} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

7.4 Results

The confusion matrices in Figure 7.1 summarise how the system classified the 137 methods in the dataset, separating the two experimental conditions (with and without the Java Call Graph) and also reporting the aggregated results.



(a) Evaluation on APKs that include a JFG

(b) Evaluation on APKs that do not include a JFG

(c) Overall evaluation across the full set of APKs

Figure 7.1: Confusion metrics for the vulnerability classification task.

These results highlight that the system maintains very low false-negative rates across all configurations. When the JCG is available, only 5 % of vulnerabilities are missed, and even without it the rate remains at just 3 %.

At the same time, the absence of Java-side context increases uncertainty. When the JCG is unavailable, the model compensates by adopting a more conservative stance, inflating the false-positive rate from 18 % to 40 %. This confirms that cross-layer information plays a critical role in helping the LLM distinguish between a vulnerability and a benign application-level fault.

Considering the full dataset, the aggregated results show a distribution of 17 % true positives, 49 % true negatives, 30 % false positives, and only 4 % false negatives. This overall pattern

reinforces the behaviour observed in both experimental settings: the system reliably identifies most real vulnerabilities but tends to over-report suspicious cases when contextual information is limited. Such characteristics make the approach well suited for an initial triage stage.

Metric	With JCG	Without JCG	All APKs
Accuracy	77.05 %	57.33 %	66.18 %
Precision	54.17 %	25.00 %	35.94 %
Recall	81.25 %	83.33 %	82.14 %
F1-Score	65.00 %	38.46 %	50.00 %

Table 7.2: Evaluation metrics computed from the classification results.

Table 7.2 summarises the overall performance of the system under the two conditions. The aggregated accuracy (66.18 %) indicates that roughly two-thirds of all classifications match the ground truth.

Accuracy is substantially higher when the Java Call Graph is available (77.05 %) than when it is not (57.33 %), showing that Java-to-native context improves the model's ability to correctly classify both vulnerable and non-vulnerable cases.

Precision shows an even more pronounced dependency on the availability of the JCG. With JCG, 54.17 % of positive predictions correspond to true vulnerabilities, whereas without it precision drops to 25.00 %. Thus, the JCG significantly reduces over-prediction and makes positive classifications more trustworthy.

Recall remains consistently high across both settings. With JCG, the system correctly identifies 81.25 % of all true vulnerabilities, and even without it recall stays at 83.33 %. This shows that the system rarely misses dangerous cases, regardless of the context provided.

The F1-score highlights the overall effect of contextual information on the balance between precision and recall. With JCG, the system reaches 65.00 %, while without it the score falls to 38.46 %. The combined F1-score across all crashes is 50.00 %, indicating that the classifier is considerably more stable and better calibrated when Java-level execution context is available, and becomes less reliable when this information is removed.

8 Discussion

8.1 Tool Performance and Efficiency

The implemented pipeline demonstrates acceptable and promising operational performance.

Single-crash analysis by the LLM requires on average less than a minute, which makes the approach scalable and significantly accelerates the classification process. However, the total execution time per application depends heavily on fuzzing procedure, the crash count of method, the complexity of native stack traces, and the number of native libraries involved. In applications with many or large native libraries, the overhead caused by importing and decompiling binaries can substantially increase the total runtime, this overhead originates from the reverse-engineering tools rather than from the analysis performed by the model.

8.2 Effect of Java Call Graph

As seen in Chapter 7.4, the comparison between analyses performed with or without the JCG yields a clear trade-off:

- When the Java Call Graph is available ("With JCG"), the model can leverage both native and Java-level context, producing more coherent and realistic vulnerability assessments. Under this configuration, the system achieves higher recall and maintains relatively few false negatives.

- Without the Java Call Graph ("Without JCG"), the lack of Java-side context leads the model to over-approximate risks. Without knowing the origin or constraints of the data, the system tends to treat values as attacker-controlled, even when in reality they may originate from fixed Java-side logic and be impossible for an attacker to influence.

8.3 Reliability

The pipeline exhibits a low number of False Negatives, indicating that genuine vulnerabilities are rarely overlooked. This behaviour is partly influenced by the LLM's tendency to classify ambiguous memory-related faults (e.g., buffer errors, JNI misuse, invalid native calls) as potentially dangerous. Prior studies highlight similar patterns: general-purpose models often over-predict positives, hallucinate tool behaviour, or misinterpret control-flow semantics [50, 57].

8.4 Limitations

The evaluation performed in this thesis is subject to some limitations that affect the results.

The quality of the classification strongly depends entirely on the crash traces produced by POIROT. When stack trace contain missing or unresolved entries (marked as "`??`"), the model lacks essential execution context and tends to assume a worst-case scenario, contributing to the high false-positive rate observed in the evaluation.

The dataset, though representative of diverse applications, cannot fully capture the variability of real-world Android ecosystems.

8.5 Comparison with Ground-Truth Vulnerability

To evaluate the quality of the LLM-based triage, this section compares the model's classification of a real vulnerability in the `tpCamera` application with the ground truth reported in the POIROT paper [7]. The vulnerability, later assigned CVE-2023-30273, concerns a use-after-free condition in the native MP4 encoding library.

8.5.1 Ground-Truth Summary

Section 5.6 of the POIROT paper [7] describes a reproducible use-after-free vulnerability in the `libTPMp4Encoder.so` library. The flaw is triggered when the application invokes the MP4

encoding pipeline with malformed JPEG or H.264 data:

1. The JNI function `packVideo` receives attacker-controlled frame data and a size parameter.

2. The native function `mp4_write_one_h264` tears down the encoder context on malformed input: it closes the file, frees multiple internal buffers, and then frees the global structure `iniPacker_global`.

3. A second invocation of `packVideo` reuses the same freed encoder context, triggering a *use-after-free* where the freed structure is overwritten by attacker-controlled data.

4. The overwritten structure corrupts the file pointer passed to `fclose()`, enabling arbitrary code execution under certain conditions.

The vulnerability has practical security impact: an attacker on the same network, or one who has compromised a TP-Link camera, can exploit the issue as soon as the user presses the "record" button in the tpCamera app.

8.5.2 LLM Classification Summary

The LLM-based triage correctly identifies the crash in `tpCamera` as a genuine vulnerability, assigning a *high* severity level and a confidence score of 0.85. The model attributes the crash to a memory-safety error in the native MP4 encoding pipeline, centred around the logic implemented in the `mp4_write_one_h264` routine in `libTPMp4Encoder.so`.

The full classification output for this application is provided in Appendix B. The analysis points to three main contributing factors:

- **Unvalidated size parameter**: The JNI method `packVideo` forwards an unbounded, attacker-influenced size directly to the native encoder without consistency checks.

- **Unsafe error-handling path**: On malformed Network Abstraction Layer (NAL) input, `mp4_write_one_h264` frees multiple internal buffers and then frees the encoder context itself, while higher-level code may continue using the same structure.

- **Potential heap corruption and double free**: Repeated calls on a freed context can corrupt heap metadata, consistent with the crash `scudo::reportInvalidChunkState`.

The model concludes that the vulnerability likely enables heap corruption or a double-free/use-after-free condition, mapping it to CWE-787 (Out-of-bounds Write), CWE-415 (Double Free), and CWE-416 (Use After Free).

It also identifies a plausible exploitation vector: `"trigger_method"`: "Malformed H.264 frame buffer passed via MP4Encoder.packVideo with inconsistent size parameter".

8.5.3 Comparison with Ground Truth

The LLM-generated analysis aligns closely with the ground-truth description reported in the POIROT paper [7]. Both sources identify the vulnerability as a use-after-free condition arising from repeated invocations of `packVideo` on a freed encoder context, although the LLM expresses the use-after-free through its CWE assignment rather than presenting it as the primary causal mechanism.

- **Root cause agreement**: The analysis matches the PIROT description, recognising that the encoder context (`iniPacker_global`) is freed on malformed input and later reused, enabling a use-after-free.
- **Impact on file handling**: The classification highlights that tearing down and reusing the encoder context affects file and stream management, which is consistent with the PIROT finding that corruption of this context can influence the `FILE*` pointer passed to `fclose()`.
- **Attack preconditions**: It correctly notes that the vulnerability is remotely triggerable through malformed video frames delivered over an untrusted network channel.
- **Code execution potential**: The analysis further infers that this memory-safety primitive can lead to arbitrary function invocation or code execution under realistic conditions.

Overall, the model captures the key structural and behavioural elements of the vulnerability, providing an analysis that is consistent with the real exploit chain reported in the ground-truth study.

8.5.4 Interpretation of Results

This comparison shows that the LLM-based triage is capable of reconstructing the essential characteristics of a complex native vulnerability using stack-trace evidence, decompiled code,

and contextual information accessed through MCP tools. The model correctly identifies the use-after-free pattern, the conditions that lead to encoder-state corruption, and the attacker-controlled inputs required to reach the vulnerable code paths.

The analysis produced by the LLM is not only consistent with the POIROT ground truth, but also captures several realistic exploitation vectors. Demonstrates that the proposed pipeline can approximate expert-level reasoning and align closely with real-world vulnerability reports, validating its effectiveness as an initial triage mechanism.

9 Future Work

This thesis has shown that an LLM equipped with MCP-based access to Jadx and Ghidra can assist in the triage of crashes originating from native libraries in Android applications. Several implementations could be developed in the future, particularly to further improve the quality of the triage.

9.1 Improving Crash Context and Input Visibility

The results suggest that LLM-based triage, when enriched with context retrieved via MCP tools, becomes significantly more informative and reliable during crash analysis.

A key limitation of the current implementation is the lack of visibility into the inputs that triggered each crash. Since POIROT does not expose the generated test cases or more detailed execution logs, the model is forced to reason solely from stack traces and decompiled code, without knowing which specific input conditions caused the failure.

Providing richer execution context, such as the exact fuzzer input, argument values or runtime logs, would enable the LLM to make more grounded and accurate classifications. Such information could help the model distinguish crashes triggered by malformed or unrealistic inputs from those that reflect genuine, exploitable vulnerabilities.

9.2 Use of Specialised Models for Triage

The current approach relies on general-purpose LLMs, guided through meta-prompts and tool use. Using models specifically adapted or trained for vulnerability analysis could improve stability, enhance recognition of recurring vulnerability patterns, and provide more consistent assessments across similar crashes as recent empirical evidence shows that fine-tuned, code-specialised models systematically outperform general LLMs across detection, assessment, and localisation tasks [51].

9.3 Persistent Agents

In the current implementation, each crash is analysed in isolation: the agent processes the crashes associated with a method, interacts with the tools, produces a report, and then its context is reset. A possible extension is to maintain a *persistent* triage agent over time. That could be preceded by a lightweight training or calibration phase using an available ground-truth labeled dataset. Such an approach would allow the agent to internalise verified classifications and use them as stable reference points in subsequent analyses.

A persistent agent could progressively accumulate knowledge about specific libraries, recurring code idioms, and previously classified vulnerabilities. For example, it could remember that a particular JNI library has already been associated with a confirmed buffer overflow and reuse this information when triaging future crashes involving the same code. It could also enforce dataset-wide consistency, ensuring that similar crash patterns across different APKs or execution traces receive comparable classifications and severity assessments.

9.4 Use of Retrieval Augmented Generation

RAG supplements an LLM with an external knowledge base that can be queried at inference time. In practice, it augments the model by retrieving relevant task-specific data before generation, rather than relying solely on the LLM's internal representations.

Recent work shows that RAG can strengthen factual grounding in vulnerability analysis. LProtector[53] improves detection accuracy and explainability by retrieving security-relevant context, while VulRAG[58] demonstrates that retrieving similar vulnerabilities or code fragments leads to more reliable classification.

So, a possible extension would be to build a repository of pre-analysed APKs, crashes, and ground-truth labels, allowing the agent to retrieve historical cases similar to the current one. Using these annotated crashes as an external reference, as ground truth and as concrete comparison, could help the agent anchor its reasoning, reduce ambiguity, and improve the quality and consistency of the classification.

9.5 GAN-like Multi-Agent Structure

Another possible extension is to explore a GAN-like architecture composed of multiple LLM agents with opposing objectives. One agent would attempt to demonstrate that a crash is exploitable, while a second agent would argue the opposite by seeking alternative explanations or benign root causes. A third, neutral agent would then act as an adjudicator, evaluating the evidence produced by both sides and issuing the final classification.

9.6 LLM-Based Exploit Generation and Validation

At the current stage, the agent is limited to analysing crashes and explaining possible root causes; it does not attempt to construct or validate concrete exploits. A natural extension would be to introduce a post-triage phase in which the LLM engages in an iterative reasoning-and-action cycle, similar to ReAct-style prompting. Given a crash, the agent would generate a candidate exploit input or harness modification, execute it in a controlled environment, observe the resulting behaviour and logs, and, if necessary, query additional code context via MCP. Based on this feedback loop, the agent could refine or redesign the exploit attempt, repeating the process until the vulnerability is confirmed as either exploitable or not.

10 Conclusions

This thesis presented an autonomous LLM-base triage pipeline, that integrates the Model Context Protocol (MCP) to analyse native crashes in Android applications. The goal, was to support vulnerability assessment in scenarios where manual crash inspection is performed, which can be time-consuming and labour-intensive, and where applications rely on native components accessed through the Java Native Interface.

The proposed system integrates two reverse-engineering tools into the LLM's reasoning loop: *Jadx*, for bytecode analysis and *Ghidra*, for native disassembly. Such process enables context-aware classification of *POIROT*-generated crashes.

The evaluation, across 137 crashes from 80 real-world applications, demonstrated that the workflow reliably identifies real vulnerabilities, achieving consistently low false-negative rates (3–5 %), with an an overall accuracy of roughly 66 %. Its performance improved substantially when Java–to–native contextual information is available through the Java Call Graph, accuracy increases to 77 % and precision more than doubles, confirming the importance of cross-layer information in reducing over-approximation.

The TP-LINK tpCamera case study, further shows that the system can approximate expert-level analysis. The LLM classification aligns with the ground-truth vulnerability reported in POIROT (CVE-2023-30273), correctly identifying the unsafe lifecycle of the encoder context and the conditions leading to a use-after-free.

At the same time, the evaluation performed by the tool highlighted several areas where the tool can be improved. Precision remains modest, due to incomplete crash traces and the conser-

vative behaviour of general-purpose LLMs tends to over-approximate risk when information is missing. Therefore, at the current stage, the tool rather than replacing traditional analyses, can serve as a useful integrative tool that accelerates classification and provides a structured starting point for further and more precise investigation.

Future work should focus on improving precision through richer static and dynamic context, extending Java–to–native call-graph reconstruction. Integrating automated exploitability assessment and symbolic reasoning layers could also further strengthen the system's reliability.

In conclusion, this thesis shows that LLM-based crash triage, when combined with reverse-engineering tools through MCP, offers a promising direction for scalable vulnerability analysis. It enhances the efficiency and consistency of the triage process and provides a practical foundation for more advanced, context-aware security automation.

References

[1] "Mobile operating system market share worldwide - september 2025," https://gs.statcounter.com/os-market-share/mobile/worldwide, accessed 2025-10-06.

[2] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, "Sok: Lessons learned from android security research for appified software platforms," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 433–451.

[3] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, "The android platform security model," *ACM Trans. Priv. Secur.*, vol. 24, no. 3, 2021. [Online]. Available: https://doi.org/10.1145/3448609

[4] M. Jurczyk, "Mms exploit part 5: Defeating android aslr, getting rce," https://googleprojectzero.blogspot.com/2020/08/mms-exploit-part-5-defeating-aslr-getting-rce.html, 2020.

[5] S. Almanee, A. Ünal, M. Payer, and J. Garcia, "Too quiet in the library: An empirical study of security updates in android apps' native code," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21. IEEE Press, 2021, pp. 1347–1359. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00122

[6] S. Mergendahl, N. Burow, and H. Okhravi, "Cross-language attacks," in *Network and Distributed System Security Symposium (NDSS)*, 2022. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2022-78-paper.pdf

[7] L. Di Bartolomeo, P. Mao, Y.-J. Tung, J. Ayala, S. Doria, P. Celada, M. Busch, J. Garcia, E. Losiouk, and M. Payer, "Hercules droidot and the murder on the jni express," in *Proceedings of the 34th USENIX Conference on Security Symposium*, ser. SEC '25. USA: USENIX Association, 2025. [Online]. Available: https://dl.acm.org/doi/10.5555/3766078.3766246

[8] "Platform architecture," https://developer.android.com/guide/platform, android Developers, accessed 2025-10-06.

[9] "Get started with the ndk," https://developer.android.com/ndk/guides, android Developers, accessed 2025-10-06.

[10] "Concepts | android ndk," https://developer.android.com/ndk/guides/concepts, android Developers, accessed 2025-10-06.

[11] "Jni tips," https://developer.android.com/ndk/guides/jni-tips.

[12] "Java native interface specification - introduction," https://docs.oracle.com/en/java/javase/21/docs/specs/jni/intro.html, oracle, accessed 2025-10-06.

[13] "Application sandbox," https://source.android.com/docs/security/app-sandbox, android Open Source Project, accessed 2025-10-06.

[14] "Security-enhanced linux in android," https://source.android.com/docs/security/features/selinux, android Open Source Project, accessed 2025-10-06.

[15] "Use of native code," https://developer.android.com/privacy-and-security/risks/use-of-native-code, android Developers, accessed 2025-10-07.

[16] P. Mao, M. Busch, and M. Payer, "{NASS}: Fuzzing all native android system services with interface awareness and coverage," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 4225–4243. [Online]. Available: https://www.usenix.org/system/files/usenixsecurity25-mao.pdf

[17] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. De Geus, C. Kruegel, G. Vigna *et al.*, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *The Network and Distributed System Security Symposium 2016*, 2016, pp. 1–15. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/going-native-large-scale-analysis-android-apps-practical-native-code-sandboxing-policy.pdf

[18] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007. [Online]. Available: https://dl.acm.org/doi/10.5555/1324770

[19] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated white-box fuzz testing," in *Proceedings of NDSS*, 2008. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/Automated-Whitebox-Fuzz-Testing-paper-Patrice-Godefroid.pdf

[20] "Hwaddress sanitizer - ndk," https://developer.android.com/ndk/guides/hwasan.

[21] "Addresssanitizer," https://source.android.com/docs/security/test/asan, android Open Source Project, accessed 2025-10-06.

[22] "Addresssanitizer — clang documentation," https://clang.llvm.org/docs/AddressSanitizer.html, lLVM Project Documentation (accessed 2025-10-08).

[23] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020. [Online]. Available: https://www.usenix.org/system/files/woot20-paper-fioraldi.pdf

[24] "Libfuzzer — a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. [Online]. Available: https://papers.neurips.cc/paper/7181-attention-is-all-you-need.pdf

[26] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, and et al., "A survey on hallucination in large language models," *arXiv preprint arXiv:2309.01219*, 2023. [Online]. Available: https://arxiv.org/abs/2311.05232

[27] S. Farquhar, J. Kossen, L. Kuhn, Y. Gal *et al.*, "Detecting hallucinations in large language models using semantic entropy," *Nature*, vol. 630, pp. 625–630, 2024. [Online]. Available: https://www.nature.com/articles/s41586-024-07421-0

[28] J. Spracklen, R. Wijewickrama, A. H. M. N. Sakib, A. Maiti, B. Viswanath, and M. Jadliwala, "Package hallucinations: How llms can invent vulnerabilities," *;login: USENIX Magazine*, June 2025, abbreviated version of a study to appear at USENIX Security 2025. [Online]. Available: https://www.usenix.org/publications/loginonline/we-have-package-you-comprehensive-analysis-package-hallucinations-code

[29] (2025) Architecture overview - model context protocol. [Online]. Available: https://modelcontextprotocol.io/docs/learn/architecture

[30] (2025) What is the model context protocol (mcp)? [Online]. Available: https://modelcontextprotocol.io/docs/getting-started/intro

[31] (2025) Transports - model context protocol. [Online]. Available: https://modelcontextprotocol.io/specification/2025-06-18/basic/transports

[32] (2025) Understanding mcp servers - model context protocol. [Online]. Available: https://modelcontextprotocol.io/docs/learn/server-concepts

[33] (2025) Understanding mcp clients - model context protocol. [Online]. Available: https://modelcontextprotocol.io/docs/learn/client-concepts

[34] "Mcp - model context protocol," https://modelcontextprotocol.io/docs/concepts/tools, 2025.

[35] "About CWE," https://cwe.mitre.org/about/index.html, mITRE.

[36] "CWE list and data," https://cwe.mitre.org/data/index.html, mITRE.

[37] "Common vulnerabilities and exposures (cve) — overview," https://www.cve.org/about/overview.

[38] "CVEs and the NVD process," https://nvd.nist.gov/general/cve-process, nIST NVD.

[39] "CVSS v4.0 official support in NVD," https://nvd.nist.gov/general/news/cvss-v4-0-official-support, nIST NVD, 2024.

[40] "Cvss v3.1 user guide (revision 1)," https://www.first.org/cvss/v3-1/cvss-v31-user-guide_r1.pdf, fIRST.

[41] Threat NG Staff. (2025) Vulnerability triage. [Online]. Available: https://www.threatngsecurity.com/glossary/vulnerability-triage

[42] T. Cipresso, "Software reverse engineering," in *Encyclopedia of Bioinformatics and Computational Biology*. Springer, 2010. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-04117-4_31

[43] P. Samuelson, R. Davis, M. D. Kapor, and J. H. Reichman, "Reverse engineering: Legal and policy issues," *Communications of the ACM*, 1990. [Online]. Available: https://people.ischool.berkeley.edu/~pam/papers/ieee_1990.pdf

[44] "Ghidra: A software reverse engineering (sre) framework," https://github.com/NationalSecurityAgency/ghidra.

[45] "jadx: Dex to java decompiler," https://github.com/skylot/jadx.

[46] E. Basic and A. Giaretta, "Large language models and code security: A systematic literature review," *arXiv preprint arXiv:2412.15004*, 2024.

[47] Y. Jiang, J. Liang, F. Ma, Y. Chen, C. Zhou, Y. Shen, Z. Wu, J. Fu, M. Wang, S. Li *et al.*, "When fuzzing meets llms: Challenges and opportunities," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 492–496. [Online]. Available: https://doi.org/10.1145/3663529.3663784

[48] Y. Liu, J. Deng, X. Jia, Y. Wang, M. Wang, L. Huang, T. Wei, and P. Su, "Promefuzz: A knowledge-driven approach to fuzzing harness generation with large language models." [Online]. Available: https://pvz122.github.io/pdf/25-promefuzz.pdf

[49] M. J. Torkamani, J. Ng, N. Mehrotra, M. Chandramohan, P. Krishnan, and R. Purandare, "Streamlining security vulnerability triage with large language models," *arXiv preprint arXiv:2501.18908*, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2501.18908

[50] V. Akuthota, R. Kasula, S. T. Sumona, M. Mohiuddin, M. T. Reza, and M. M. Rahman, "Vulnerability detection and monitoring using llm," in *2023 IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE)*, 2023, pp. 309–314.

[51] X. Yin, C. Ni, and S. Wang, "Multitask-based evaluation of open-source llm on software vulnerability," *IEEE Transactions on Software Engineering*, vol. 50, no. 11, pp. 3071–3087, 2024.

[52] U. Kulsum, H. Zhu, B. Xu, and M. d'Amorim, "A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback," in *Proceedings of the 1st ACM International Conference on AI-Powered Software*, ser.

AIware 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 103–111. [Online]. Available: https://doi.org/10.1145/3664646.3664770

[53] Z. Sheng, F. Wu, X. Zuo, C. Li, Y. Qiao, and L. Hang, "Lprotector: An llm-driven vulnerability detection system," 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2411.06493

[54] Y. Zhang, Y. Yuan, and A. C.-C. Yao, "Meta prompting for ai systems," 2025. [Online]. Available: https://arxiv.org/abs/2311.11482

[55] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023. [Online]. Available: https://arxiv.org/abs/2201.11903

[56] "Ibm: nm command," https://www.ibm.com/docs/en/aix/7.3.0?topic=n-nm-command.

[57] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, "Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks," 2024. [Online]. Available: https://arxiv.org/abs/2312.12575

[58] X. Du, G. Zheng, K. Wang, Y. Zou, Y. Wang, W. Deng, J. Feng, M. Liu, B. Chen, X. Peng, T. Ma, and Y. Lou, "Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag," 2025. [Online]. Available: https://arxiv.org/abs/2406.11147

A

Appendix A

The following listing shows the system prompt used for the vulnerability triage.

```
You are a **senior mobile reverse-engineering & security engineer**.
You will receive **one CrashEntry at a time** from a JNI-fuzzing triage pipeline.
Your task is to decide whether the crash is **LIKELY caused by a genuine code vulnerability** (memory safety, logic bug, or exploitable condition) **or NOT** (e.g., harness/environmental issue, non-exploitable crash, or benign failure).
Return **ONLY** a single JSON object that strictly follows the schema below.

---

## 1. Definition of a *vulnerability*
A crash is a **vulnerability** if, and only if, there is clear code-level evidence that program state or control flow can be —influenceddirectly or —indirectlyby data that an attacker can realistically affect,
and that this influence can lead to unsafe behaviour such as memory corruption, information leakage, control-flow hijacking, or a meaningful denial-of-service under realistic conditions.
The attacker rarely has *complete* control of every value. Therefore the assessor MUST reason backward from the crash site through the calling
```

```
    chain to determine whether the values involved (parameters, lengths,
    indexes, flags) originate from attacker-influenced inputs or from fixed/
    validated assignments earlier in the call path.
If an intermediate function performs validation, transformation, or enforces
    fixed values, that reduces (or removes) attacker control and must be noted
    .

Concrete examples that qualify as *vulnerable*:
- Out-of-bounds read/write
- Integer overflow affecting allocation or buffer size
- Use-after-free or double free
- Attacker-controlled format strings
- Unchecked pointer dereference in reachable code

Crashes that **should NOT** be labeled as vulnerable include:
- Infinite loops or benign aborts in code that is clearly not influenced by
    attacker-controlled input (pure functional bugs).
- Local Denial of Service (app crash) clearly caused ONLY by harness/
    environment misuse (e.g., impossible JNI stubs, intentionally passing NULL
     where production code never would).

- Harness or environment faults (e.g., NULL passed by harness, invalid
    ownership)
- Crashes under unrealistic or malformed inputs not reachable from the app
- Sanitizer/allocator aborts without supporting unsafe app-level code evidence

---

## 2. Input fields you will receive
- `process_termination`: e.g., "SIGSEGV", "abort", "ASAN: heap-use-after-free"
- `stack_trace`: list of frames or raw text
- `java_callgraph`: list of →JavaJNI call-path strings showing how Java
    execution reaches the JNI method that calls the native function involved
    in the crash; each element is formatted as "<caller> -> <callee>" and
    ordered from Java entrypoint to the JNI call.
- `app_native_function`: string or null
```

```
- `jni_bridge_method`: string or null
- `fuzz_harness_entry`: string or null
- `program_entry`: string or null
- Map of relevant libraries and their JNI methods

---

## 3. Tools and some actions
You have **Jadx MCP** and **Ghidra MCP**. You MUST use them proactively to resolve missing context.
**Do NOT stop analysis just because a function is a "wrapper" or "thunk".**

**Exploration Rules:**
1.  **Resolve Thunks/Imports:** If the crash is in a wrapper, you MUST search for the caller function in the provided `LibMap`. Decompile the CALLER to see what arguments it passes.
2.  **Cross-Library Search:** If a symbol is missing in one `.so`, look at the `LibMap` to see if it's exported by another `.so`. Use `list_functions` or `search_functions` on related libraries.
3.  **JNI Root Analysis:** Always decompile the **App Native Function** (the JNI entry point). The vulnerability often lies in how the JNI entry point parses arguments before passing them to the crashing utility function.
4.  **Java Context:** Use Jadx to check the `jni_bridge_method`. If Java passes a byte array, check if the length is validated in Java before the JNI call.

### MCP Exploration
For each crash, the you MUST use MCP tools (Ghidra + Jadx) in the following exact order:

1. Identify the FIRST application-level native frame BELOW allocators/sanitizers.
   - Examples of allocator frames to skip: scudo::*, malloc_postinit, abort, std::terminate.

2. GHIDRA MCP:
```

```
   (a) Decompile the function corresponding to that frame.
   - use `search_functions_by_name`, that return `<function> @ <addr>`, you have to use exact that address in `decompile_function_by_address <addr>`
   - Decompile all the functions address returned, using decompile_function_by_address <addr>`, one could be a wrapper/thunk.
   (b) Locate any calls to memcpy/memmove/ks_memcpy or indirect function pointers.
   (c) For each call: extract SOURCE, DESTINATION, LENGTH expressions.

3. BACKWARD DATA-FLOW:
   For each of the three arguments (src, dst, len):
      - Trace the argument backwards within the function.
      - If it comes from the caller, decompile the caller through MCP.
      - Continue recursively up to:
          - the JNI entry point, or
          - the first point where the value becomes constant or validated.

4. JNI AND JAVA ANALYSIS:
   After reaching the JNI layer:
      - Use Jadx MCP to inspect how Java constructs the arguments.
      - Determine whether LENGTH or POINTERS are attacker-controlled.
      - Determine whether any Java or JNI validation limits the effective size.

5. FUNCTION-POINTER IMPLEMENTATION CHECK:
   If the function is an indirect call (e.g., PTR_xxx):
      - Search xrefs to the function pointer.
      - Attempt resolving the implementation in the same library.
      - If missing, you MUST explicitly state it is missing (do NOT assume behavior).

6. Only AFTER steps 1-5 are complete or explicitly IMPOSSIBLE:
      → Produce classification and vulnerability judgment.

---
```

```
## 4. Analysis checklist
1. Correlate termination reason with app-level code evidence (e.g., allocator abort + unsafe `memcpy` call).
2. Look for unsafe operations: unchecked `memcpy`, pointer arithmetic, missing bounds check, double free, null deref.
2a. Backward data-flow / taint reasoning:
- Start from the crashing instruction / top native frame (e.g., `byte_array_to_bson_string`). Trace backward through callers (decompile each caller up to a reasonable depth, e.g., 3 levels) to find where the relevant variable(s) are assigned. Start from the last stack trace element, and go up following the stack trace list.
- At each step, record whether the value is: (A) directly taken from fuzzer/JNI input, (B) derived from input but transformed/checked (describe transformation), (C) set to a fixed/constant value, or (D) obtained from an environment/resource not attacker-controlled.
- If any function on the backward path performs validation (bounds checks, length checks, canonicalisation, ownership checks), note it and reduce confidence accordingly.
- When the backward path reaches a JNI bridge, query Jadx, using `java_callgraph` to orientate and inspect the Java code that constructs the native call arguments and determine whether those arguments can be influenced by untrusted sources (e.g., network input, user-supplied file, IPC payload). Record findings in `evidence` with precise snippets or references.
- If no realistic taint path from attacker-controlled sources exists, classify as non-vulnerability (Env/Harness) or at most low-confidence vulnerability and explain which assignments prevented exploitability.
- You have to analise all the providerd .so methods, and the Java call graph
3. Evaluate reachability: could untrusted input trigger this path under real app use?
4. Mark **"Env/Harness"** when crash originates from unrealistic or harness-only behavior.

---

## 5. Confidence & severity guidance
```

```
- **confidence** in [0.0, 1.0]
  - >= 0.9 → clear code-level proof of vulnerability
  - 0.6-0.8 → likely, but not fully confirmed
  - 0.3-0.5 → unclear or speculative
  - < 0.3 → unlikely or unsupported
- **severity** (only if justified by evidence):
  - `critical`: remote code execution or major compromise
  - `high`: memory corruption or data leak with realistic trigger
  - `medium`: limited DoS or local issue under complex input
  - `low`: minor or constrained condition

---

## 6. Output schema (strict JSON, no prose outside)
Return a JSON object with:
- `chain_of_thought`: strings. Write a step-by-step internal monologue BEFORE classifying.
- `is_vulnerable`: boolean. True if the crash is a vulnerability, false otherwise
- `confidence`: float (0.0-1.0)
- `reasons`: list of short bullet strings
- `cwe_ids`: list (e.g., ["CWE-787"]) or empty
- `severity`: one of ['low','medium','high','critical'] or null
- `app_native_function`: string or null
- `jni_bridge_method`: string or null
- `stack_trace`: list (normalized)
- `affected_libraries`: list of filenames or empty
- `evidence`: list of objects `{ "function": str|null, "address": str|null, "file": str|null, "snippet": str|null, "note": str|null }`
- `call_sequence`: list of strings
    Each element of the list is a Path
    Ordered list of functions (names or "name @ addr") representing the →callercallee path
    that leads from JNI/fuzzer entrypoint to the vulnerable function.
    MUST be derived through MCP cross-reference analysis.
- `recommendations`: list of short, actionable next steps
```

```
- `assumptions`: short list of assumptions
- `limitations`: short list of missing or uncertain factors

- `exploit`: null OR an object with:
    - `exploitability`: string ('unknown','theoretical','practical')
    - `trigger_method`: string or null
    - `prerequisites`: list of strings
    - `exploit_pipeline`: list of strings
    - `poc_commands`: list of strings
    - `poc_files`: list of strings
    - `notes`: string or null

## 6a. Exploit field requirements (only when is_vulnerable=true)

When a crash is classified as a real vulnerability:

1. You MUST provide an `exploit` object with concrete, realistic details.
2. The `exploit_pipeline` MUST describe, in 3-5 ordered steps, the conceptual flow an attacker would follow to exploit the vulnerability, combining prerequisites, payload preparation, triggering mechanism, and expected effect.
3. `poc_commands` MUST include at least one actionable Proof-of-Concept command
   usable on an Android device (e.g., ADB, am start, input file triggering).
4. PoC commands must be based on the available evidence:
   - If the vulnerability is triggered by malformed file input, provide commands such as
        "adb push crafted.bin /sdcard/Download/payload.bin"
        "adb shell am start -n <package>/<activity> --es file /sdcard/Download/payload.bin"
   - If triggered through an exported component, produce a realistic `am start ` line.
   - If the vulnerability is inside a JNI call reachable from Java, reconstruct the simplest
     feasible invocation path consistent with the java_callgraph.
5. Never fabricate missing fields: if trigger path, activity name, or
```

```
    filenames are unknown,
   include placeholders (e.g., "/sdcard/Download/payload.bin") and state
    assumptions
   in the `assumptions` field.

Rules:
- If `is_vulnerable == true`, the `exploit` field MUST be present and non-null
    .
- If `is_vulnerable == false`, the `exploit` field MUST be `null`.
- Never invent values. Use null or [] when unknown.
- Confidence must reflect actual certainty.
- Keep all text concise (max 1-3 short items per list).
```

Listing A.1: Full system prompt used by the LLM-based vulnerability triage pipeline.

B

Appendix B

The following listing shows the `report.json` file, generated by this project for the classification of the crash observed in the application `com.tplink.skylight`.


```
{
  "analysis": {
    "tool": {
      "model_name": "gpt-5.1",
      "apk_path": "APKs/com.tplink.skylight/base.apk",
      "version": "1.0"
    },
    "app": {
      "app_name": "tpCamera",
      "package": "com.tplink.skylight",
      "min_sdk": 21,
      "target_sdk": 30,
      "version_name": "3.1.20",
      "version_code": "388"
    },
    "analysisResults": [
      {
        "crash": {
          "ProcessTermination": "abort",
```

```
        "StackTrace": [
          "abort",
          "scudo::die",
          "scudo::ScopedErrorReport::~ScopedErrorReport",
          "scudo::reportInvalidChunkState",
          "scudo::Allocator<scudo::AndroidConfig, &scudo_malloc_postinit>::
deallocate",
          "mp4_write_one_h264",
          "Java_com_tplink_skylight_common_jni_MP4Encoder_packVideo",
          "fuzz_one_input",
          "main"
        ],
        "JNIBridgeMethod": "<com.tplink.skylight.common.jni.MP4Encoder:
packVideo([BIJ)I>",
        "JavaCallGraph": [
          "<com.tplink.skylight.common.manage.multiMedia.encoder.
H264_AAC_Mp4Encoder$2: run()V> -> <com.tplink.skylight.common.jni.
MP4Encoder: packVideo([BIJ)I>"
        ],
        "FuzzHarnessEntry": "fuzz_one_input",
        "ProgramEntry": "main",
        "LibMap": {
          "APKs/com.tplink.skylight/lib/arm64-v8a/libTPMp4Encoder.so": [
            "mp4_write_one_h264",
            "Java_com_tplink_skylight_common_jni_MP4Encoder_packVideo",
            "Java_com_tplink_skylight_common_jni_MP4Encoder_packVideo"
          ]
        }
      },
      "assessment": {
        "chain_of_thought": [
          "Top native frame below allocator is mp4_write_one_h264 in
libTPMp4Encoder.so. Scudo abort indicates invalid free or double free.
Need to see how mp4_write_one_h264 manages allocations and frees and
whether it can be called multiple times with same context or with
corrupted state from fuzz input.",
```

```
48          "Decompiled mp4_write_one_h264(char *param_1,int param_2,int
    param_3,undefined8 *param_4) shows parsing of H.264 NAL units from param_1
     buffer. It checks parameter validity partially: param_4 and param_1 non-
    null, __stream from *param_4 non-null, param_2>3, param_3>=0, and 0
    x00000001 start codes in first 3-4 bytes. It then examines NAL type (bVar1
     & 0x1f). For SPS (7) and PPS (8) it conditionally malloc a buffer if
    corresponding length field at param_4+6/8 is 0, assign pointer at param_4
    [5]/[7], and memcpy(__dest,__src,__size) where __size = param_2 - iVar5.
    Here length and source size are controlled by param_2 and param_1 content.
     However memcpy length is equal to malloc size, so not OOB. The bug is
    likely in lifecycle management of param_4 and its fields when error path
    triggers.",
49          "On error, function goes to LAB_001053f8: fclose(__stream); then
    LAB_00105400: if (param_4[1]) fclose(); then free(param_4[5]); free(
    param_4[7]); FUN_00104e50(param_4); MP2_encode_close(param_4[0x15]); free(
    param_4); return -1. This tears down the entire encoder context and frees
    internal buffers plus context. If mp4_write_one_h264 is subsequently
    called again with same param_4 pointer by higher-level code, that would
    cause double-free or invalid free in allocator, matching scudo::
    reportInvalidChunkState. The fuzz harness likely reuses encoder context
    after a failure; need to see JNI wrapper for how param_4 is obtained and
    whether it's stored globally.",
50          "JNI entry
    Java_com_tplink_skylight_common_jni_MP4Encoder_packVideo takes (JNIEnv*,
    jobject, jbyteArray, jlong, jint size, jint something). Decompiled code:
    it validates param_4 (size) only for negative, logs error, else logs video
     size, calls virtual function at (*param_1 + 0x5c0)(param_1,param_3,0) to
    get jpg buffer (lVar2). If lVar2==0 log failure. Else if global
    DAT_0011e6a8==0 error; else uVar1 = mp4_write_video(lVar2,param_4,param_5)
    . mp4_write_video presumably wraps mp4_write_one_h264 with encoder context
     stored globally or in param_1/DAT_0011e6a8. JNI always calls free
    callback at (*param_1+0x600)(param_1,param_3,lVar2,0), so buffer ownership
     is well defined.",
51          "We 'dont have mp4_write_video here, but mp4_write_one_h264
    signature shows param_4 is an encoder context that contains stream handles
     and allocated SPS/PPS buffers. That context must be created elsewhere (
```

```
likely in an init JNI). Critical point: mp4_write_one_h264 itself decides
to fully tear down encoder context (closing file and freeing param_4) on
any malformed NAL (e.g., bad header) and then returns -1. If caller
continues using same handle or global, it will cause UAF or double free.
However, responsibility for not reusing freed context is in higher-level
code. We 'dont yet see evidence in this crash that mp4_write_one_h264 is
called twice on same param_4; the allocator abort could instead be from
freeing param_4[5]/[7] after corruption via out-of-bounds write. Need to
re-examine memcpy and size computations for potential mismatch or integer
issue.",
52          "In SPS/PPS case, they set *(int*)(param_4+6)=param_2-iVar5 then
__size=(size_t)(param_2-iVar5); __dest=malloc(__size); param_4[5]=__dest;
then memcpy(__dest,__src,__size). __src points into param_1 buffer
immediately after start code. 'Theres no bounds check on param_1 but
memcpy length equals allocation, and both derived from param_2 (packet
size). If caller passes param_2 inconsistent with real buffer length, 'its
 UAF/overflow at the Java/native boundary: mp4_write_one_h264 trusts
param_2. We need to see how mp4_write_video supplies param_2 and whether
it uses fuzzer-controlled size from packVideo (param_4 jint).",
53          "JNI packVideo passes size (param_4) directly into mp4_write_video
 without additional checks beyond size>=0. There is no upper bound or
consistency check vs actual byte array length. In Java, packVideo is
invoked from H264_AAC_Mp4Encoder$2.run() with a byte[] and an int length.
We lack Java code but fuzz harness likely mutates both array length and
size parameter arbitrarily. In real app, size probably equals buffer
length; but fuzzer may pass inconsistent pair causing mp4_write_video/
mp4_write_one_h264 to read or write beyond buffer, leading to heap
corruption and allocator abort. However, our decompiled native stack for
mp4_write_one_h264 shows memcpy only writes inside its own allocated
buffer; reading from param_1 can be OOB if param_2>actual buffer length,
but we don't see a crashing read in this frame; allocator detects
corrupted chunk later on free.",
54          "Key question: Is param_1 pointer and its backing allocation sized
 according to size (param_2), or is there a trusted data source? From JNI,
 lVar2 buffer from virtual callback (**(code **)(*param_1 + 0x5c0)) is
likely of some internal camera frame size independent of 'size'. If
```

```
      mp4_write_video passes (lVar2, size, ...) and size>frame buffer length,
      mp4_write_one_h264 will read beyond lVar2, corrupting heap or causing
      invalid free. That is a genuine native vulnerability reachable from
      attacker-influenced size parameter (through Java). On the other hand, if
      virtual callback always allocates at least size bytes or uses size as
      allocation parameter, then app is consistent. Hard to tell without
      mp4_write_video and that callback. But presence of separate 'size' arg
      strongly suggests risk if not validated.",
55          "scudo::reportInvalidChunkState typically indicates heap metadata
      corruption or double free. mp4_write_one_h264 frees several pointers:
      param_4[5], param_4[7], param_4, and also calls FUN_00104e50 which likely
      frees additional fields. If some of those pointers were already freed
      earlier (e.g., on previous error), double free occurs. However,
      mp4_write_one_h264 only frees them on error path or when param_4/param_1
      null; on success or NAL types not 7/8/1/5, it just enqueues frame and
      writes to file. The error path is reachable based solely on input data (
      NAL header) and param values; it closes stream and tears down context then
       returns. No apparent guard to prevent subsequent calls with same param_4;
       that would make subsequent mp4_write_one_h264 hits call fclose/free on
      invalid pointers, causing allocator abort. Whether fuzzer replicates that
      pattern depends on harness, but in real-world use, caller may continue to
      reuse encoder after encountering malformed frame from network, causing
      crash.",
56          "Given the fuzz context, jni_bridge uses untrusted size, and
      mp4_write_video likely passes it down; mp4_write_one_h264 reads param_1
      buffer based on param_2 size without checking against actual buffer
      capacity, and tears down context with multiple frees on error. Both
      patterns are unsafe. Without mp4_write_video we can't prove exact heap
      corruption mechanism, but the combination of scudo invalid chunk state,
      multi-free on global context, and missing size validation linking Java
      size argument to native buffer means this is a likely genuine
      vulnerability. It is not a harness-only artifact; Java API exposes
      packVideo which could be called with inconsistent size from untrusted
      media source inside the app.",
57          "Classifying as vulnerable (memory safety, heap corruption /
      double free). Severity high: crash and potential RCE if attacker can shape
```

```
 NAL stream and lengths; confidence moderate-high (~0.75) due to indirect
evidence and missing helper function code. CWE candidates: CWE-787 (out-of
-bounds read/write) and CWE-415 (double free)."
      ],
      "is_vulnerable": true,
      "confidence": 0.85,
      "reasons": [
        "mp4_write_one_h264 trusts size parameter (param_2) to describe
buffer at param_1 without bounds checking; size can be influenced via Java
 packVideo size argument",
        "On error path, mp4_write_one_h264 closes and frees encoder
context param_4 and internal buffers, but higher-level code may still
reuse context, leading to double free or invalid free detected by scudo",
        "JNI entry packVideo performs almost no validation and passes
attacker-influenced size directly to mp4_write_video/mp4_write_one_h264"
      ],
      "cwe_ids": [
        "CWE-787",
        "CWE-415",
        "CWE-416"
      ],
      "severity": "high",
      "affected_libraries": [
        "libTPMp4Encoder.so"
      ],
      "evidence": [
        {
          "function": "
Java_com_tplink_skylight_common_jni_MP4Encoder_packVideo",
          "address": "0010b588",
          "file": "libTPMp4Encoder.so",
          "snippet": "if (param_4 < 0) { ... } else { ... uVar1 =
mp4_write_video(lVar2,param_4,param_5); }",
          "note": "JNI passes unbounded size param_4 from Java directly to
 native video writing pipeline."
        },
```

```
          {
            "function": "mp4_write_one_h264",
            "address": "00105308",
            "file": "libTPMp4Encoder.so",
            "snippet": "__size = (size_t)(param_2 - iVar5);\n__dest = malloc
(__size);\n...\nmemcpy(__dest,__src,__size);",
            "note": "memcpy size derived entirely from param_2 with no
independent verification of the backing buffer at __src."
          },
          {
            "function": "mp4_write_one_h264",
            "address": "00105308",
            "file": "libTPMp4Encoder.so",
            "snippet": "fclose(__stream);\nif ((FILE *)param_4[1] != (FILE
*)0x0) fclose((FILE *)param_4[1]);\nif ((void *)param_4[5] != (void *)0x0)
 free((void *)param_4[5]);\nif ((void *)param_4[7] != (void *)0x0) free((
void *)param_4[7]);\nFUN_00104e50(param_4);\nMP2_encode_close(param_4[0x15
]);\nfree(param_4);",
            "note": "Error path frees multiple context fields and the
context itself; repeated use can cause double free/invalid chunk state."
          }
        ],
        "call_sequence": [
          "fuzz_one_input ->
Java_com_tplink_skylight_common_jni_MP4Encoder_packVideo @ 0010b588 ->
mp4_write_video (addr unknown) -> mp4_write_one_h264 @ 00105308"
        ],
        "recommendations": [
          "In mp4_write_video/mp4_write_one_h264, ensure the size argument
is derived from or clamped to the actual buffer length; add explicit
bounds checks before parsing NAL headers or copying data",
          "Remove responsibility for freeing the encoder context from
mp4_write_one_h264; instead, centralize teardown logic and ensure a
context is freed exactly once",
          "In Java packVideo, validate that the provided length does not
exceed the byte array length and add sanity upper bounds for expected
```

```
    frame sizes"
105       ],
106       "assumptions": [
107         "mp4_write_video forwards the size parameter from packVideo
    directly or with minimal transformation to mp4_write_one_h264",
108         "The buffer provided via lVar2 and the size argument can be
    inconsistent under fuzzing or malformed inputs",
109         "Encoder context param_4 is reused across multiple
    mp4_write_one_h264 calls by higher-level code"
110       ],
111       "limitations": [
112         "Implementation of mp4_write_video is not available, so exact data
     flow for size and buffer is inferred",
113         "Java-side H264_AAC_Mp4Encoder$2.run() implementation is not
    decompiled, so exact origins of packet length and data are unknown",
114         "Crash log does not show instruction address of failure;
    classification is based on allocator abort plus code inspection"
115       ],
116       "exploit": {
117         "exploitability": "theoretical",
118         "trigger_method": "Malformed H.264 frame buffer passed via
    MP4Encoder.packVideo with inconsistent size parameter",
119         "prerequisites": [
120           "Ability to influence H.264 NAL stream and corresponding size
    argument passed to MP4Encoder.packVideo within the app",
121           "Target app using MP4Encoder with untrusted or semi-trusted
    video sources (e.g., network streams, remote camera feeds)",
122           "Repeated use of encoder context after an error, leading to
    double free/heap corruption"
123         ],
124         "exploit_pipeline": [
125           "Prepare H.264 frame data where the Java-visible length argument
     is larger than the actual native buffer or causes malformed NAL headers
    triggering the error path",
126           "Cause the app to call MP4Encoder.packVideo(data, len, ts) with
    this crafted payload, propagating size to mp4_write_video/
```

```
    mp4_write_one_h264",
127             "Within mp4_write_one_h264, trigger out-of-bounds reads/writes
    or context teardown with free() on internal buffers and encoder context",
128             "Induce a subsequent call to the encoder using the same (now
    freed) context, causing scudo-detected invalid free or exploitable heap
    corruption"
129           ],
130           "poc_commands": [
131             "adb push crafted_frame.bin /sdcard/Download/payload.bin",
132             "adb shell am start -n <package>/<activity> --es h264_path /
    sdcard/Download/payload.bin --ei frame_len 1048576"
133           ],
134           "poc_files": [
135             "/sdcard/Download/payload.bin"
136           ],
137           "notes": "Exact activity/class names and how to feed raw H.264
    frames into MP4Encoder.packVideo depend on the 'apps wiring; tester should
     identify the entry point where video frames are obtained and lengths set,
     then adjust PoC intent/inputs accordingly."
138         }
139       },
140       "statistics": {
141         "time": "00:00:35",
142         "llm_requests": 3,
143         "llm_tool_calls": 5,
144         "input_tokens": 21012,
145         "output_tokens": 3190
146       }
147     }
148   ]
149 }
150 }
```


Listing B.1: Example of the tool's output for the APK `com.tplink.skylight`