SAPIENZA
UNIVERSITÀ DI ROMA

Sapienza University of Rome

Dipartimento di Ingegneria Informatica, automatica e gestionale
National PhD in Artificial Intelligence for Security and Cybersecurity

THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Artificial Intelligence for Android Stealth-Attack Detection: A Digital Forensics Approach

Thesis Advisor
Prof. Giorgio Giacinto

Thesis Co-Advisor
Prof. Alessio Merlo

Candidato
Silvia Lucia Sanna
2064684

Academic Year: MMXXV-MMXXVI (XXXVIII cycle)

Copyright © by Silvia Lucia Sanna. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

A Paola e A Lello

Abstract

Android is the most popular Operating System for mobile devices worldwide due to its low cost and open-source platform. Various apps for different services have been developed, but the incorrect management of specific data structures and code sections can lead to vulnerabilities, allowing malware to spread and increasing the risks of cyberattacks. Among the vulnerabilities in Android applications, one of the most interesting is those related to the native code, i.e. C/C++ libraries used to interact with native activities and components such as camera, microphone, elaborate pictures, and fast data processing. These vulnerabilities, imported from common and popular third-party libraries or introduced by developers, refer to common C/C++ vulnerabilities, such as buffer overflow and format string vulnerabilities. When these vulnerabilities are exploited, an attacker can have access to the main memory where data is stored in clear text e.g. encryption keys. Over the years, various static and dynamic analysis techniques (both without and with execution) have been developed, particularly automatic detection systems based on Artificial Intelligence (AI) algorithms. Despite this, malware with anti-analysis and evasion techniques has been developed, for example, involving the use of obfuscation, steganography, or adversarial attacks on AI systems. For this reason, this thesis first introduces a methodology based on AI algorithms to detect and exploit risky vulnerabilities in the native code of Android applications. Secondly, a new detection mechanism based on memory forensics is presented, also resistant to common anti-analysis and adversarial Android samples. Finally, it explains how AI can be applied to Digital Forensics (DF) investigations and the importance of accurate and robust AI-based DF tools.

Keywords: Android malware, Android vulnerabilities, Memory Forensics, Android Stegomalware, Rooting Android, AI-based Digital Forensics, Android forensics

Contents

List of Figures vi

List of Tables viii

Acronyms x

1 Introduction 1

2 Contributions of this thesis 7

3 Technical Overview of Android System and Digital Forensics 9
3.1 Android System 9
3.1.1 Android OS 9
3.1.2 Android Application Package 13
3.2 Digital Forensics 18
3.3 Android Forensics 23

4 Current Research and Approaches on Android Malware Detection 25
4.1 Android Malware Detection 25
4.2 Android Forensics for Threat Analysis 29
4.2.1 Android Memory Forensics for Malware Analysis 30
4.3 Artificial Intelligence for Android Malware Forensics 32

5 Evaluating the Exploitability of Android Native Code 33
5.1 Assessing Native Code Vulnerabilities 33
5.2 Reaching Native Code Vulnerabilities 40
5.3 Exploiting Native Code Vulnerabilities 44

6 Memory Acquisition in Android Devices 48
6.1 Root Problem 48
6.2 Memory Forensics Acquisition 55
6.2.1 Overview on Android Memory Acquisition 56
6.2.2 AndroMemDump - Full RAM 56
6.2.3 AndroMemDump - Process RAM 60

7 Detecting Android Malware through Memory Analysis 64
7.1 Memory Analysis with Image Encoding 64
7.2 Memory Analysis with Audio Encoding 71
7.3 Memory Analysis with Text Encoding 76
7.4 Memory Analysis with Volatility Report 78
7.5 Detecting Android Stegomalware via Memory Forensics 82

8 Applying Artificial Intelligence to Digital Forensics **89**
8.1 Using AI in Digital Forensics 89
8.2 Evaluating the Robustness of AI-based Digital Forensics Tools 92
8.3 Digital Forensics Analysis of AI-generated content 97

9 Conclusions **100**

Bibliography **102**

List of Figures

1.1 Thesis Pipeline 6

3.1 Android OS 10
3.2 Android Package (APK) Structure 14
3.3 Digital Forensics Phases 19

5.1 Risk Estimation Pipeline 36
5.2 Risk and Librarian comparison 39
5.3 Vulnerability risk in 2023 on popular APK 40
5.4 Reachability Pipeline 43

6.1 MoLIFE Pipeline 49
6.2 MoLIFE Evaluation: DF acquisition comparison from mDT to real Android device . 53
6.3 MoLIFE retrieved one-shod Telegram picture 54
6.4 Full RAM acquisition methodology pipeline 58
6.5 Target process RAM acquisition pipelie 61

7.1 Fridump analysis pipeline 65
7.2 Full RAM analysis pipeline 65
7.3 Comparison of target-process and full RAM analysis 65
7.4 Fridump RAM analysis via image encoding 66
7.5 Confusion Matrix of Different Memory Areas using 2D images with pre-trained ResNet18 68
7.6 Feature Extraction classification and FSL memory image encoding 68
7.7 Feature Extraction classification for memory image encoding 69
7.8 Training performance image classification 70
7.9 Target-process RAM analysis via image encoding comparison with other state-of-the-art tools 71
7.10 Fridump RAM analysis via audio encoding 72
7.11 Fridump RAM analysis via text encoding 77
7.12 Full RAM analysis via Volatility and text encoding compared with Fridump string analysis 79
7.13 RAM text analysis 81
7.14 RAM Text encoding analysis 82
7.15 Android Stegomalware Thread Model 1 83
7.16 Android Stegomalware Threat Model 2 85
7.17 Percentage of APKs featuring the `Bitmap` loader variant for which VT marked with the specific tag. 86
7.18 Percentage of APKs featuring the `openCV` loader variant for which VT marked with the specific tag. 86
7.19 Results on Android Stegomalware Threat Model 2 86
7.20 Tampered real application 87
7.21 Custom tampered application 87

7.22 Android Stegomalware Threat Model 2 app runtime behavior 87

8.1 Evaluation accuracy and robustness of AI-based DF tools 94
8.2 Robustness evaluation on deepfakes in AI-based DF tools 97

List of Tables

5.1 Risk Matrix . . . 38
5.2 Native Code Contributions . . . 47

6.1 Memory Forensics Contributions . . . 63

7.1 Most significant image representation . . . 69
7.2 Classification results on static sonification . . . 74
7.3 Most significant audio representation regions . . . 75
7.4 Most significant audio encoding . . . 75
7.5 Classification details in audio encoding . . . 76
7.6 Comparison of Volatility 2 and 3 Linux Plugins . . . 80
7.7 General-purpose LLMs for Memory Analysis . . . 81
7.8 IoC list from VT and LLM analysis . . . 82
7.9 Results for Android Stegomalware Threat Model 1 . . . 84
7.10 Memory Analysis Contributions . . . 88

8.1 Applying AI to all DF main phases . . . 90
8.2 Results on the use of general-purpose LLMs for steganography activities . . . 92
8.3 Results on morphed and deepfakes in AI-based DF tools . . . 96
8.4 Face recognition evaluation on AI-based DF tools . . . 97
8.5 Artificial Intelligence and Digital Forensics Contributions . . . 99

Acronyms

ABI Application Binary Interface

ADB Android Debug Bridge

AI Artificial Intelligence

AOT Ahead-of-Time

APK Android Package

ART Android Runtime

CNN Convolutional Neural Network

CTI Cyber Threat Intelligence

CVE Common Vulnerability Exposure

CVSS Common Vulnerability Scoring System

DF Digital Forensics

DL Deep Learning

DT Digital Twin

DVM Dalvik Virtual Machine

ELF Executable and Linkable format

genAI Generative Artificial Intelligence

JIT Just-in-Time

JNI Java Native Interface

JVM Java Virtual Machine

LLM Large Language Models

mDT mobile Digital Twin

ML Machine Learning

NDK Native Development Kit

NLP Natural Language Processing

OS Operating System

VM Virtual Machine

xAI Explainable AI

Chapter 1

Introduction

Android smartphones are the most used mobile devices worldwide due to their affordability, hardware diversity, and open software ecosystem i.e. its source code is publicly available and modifiable through the Android Open Source Project (AOSP). Consequently, Android applications (APKs) represent the most common type of mobile software and the community of developer is composed of over six million active Android developers worldwide, contributing to a vast and heterogeneous app ecosystem. Notably, Android is not limited to smartphones but also smart TVs, tablets, wearable devices, and in-vehicle infotainment systems. Hence, the diffusion of Android applications depends not only on the number of users but also on the number and heterogeneity of devices. According to recent statistics, Android holds over 70% of the global mobile operating system market share [191], and it is estimated that, on average, each person owns at least one Android-powered device [89].

For these reasons, the exposure to cyberattacks such as malware, phishing, and data breaches is significantly higher on Android devices. However, this openness also facilitates malicious modifications, such as cracked or repacked applications that unofficially distribute premium features for free. Although app repacking is technically complex [137, 166, 167], it remains feasible and represents a strictly illegal activity both in terms of possession and distribution. Hence, it is crucial to analyze Android APKs before installation or release on the market to ensure that users are protected. APK analysis can be performed in three main ways: *static* (i.e. by analysing the APK structure, its code, specific files, API call, CFG without execution) [35, 157], *dynamic* (i.e. by running the application and capturing specific runtime behaviors such as called functions and parameters, dynamic CFG, network traffic, memory, hooking functions) [49, 113], or *hybrid* (i.e. combining static with dynamic) [36, 198]. Over the years, several detection techniques have been developed, many of which rely on Artificial Intelligence (AI) algorithms employing traditional Machine Learning (ML) or Deep Learning (DL) models [95, 162]. Some approaches even encode malware into human-sensorial data (i.e. visual, auditory, or textual) such as images, audio, or text (i.e. strings), enabling AI models to perform classification more effectively [145, 146]. However, current developed methodologies can be applied only after some feature engineering, i.e. a minimum analysis is required to focus on specific malicious patterns and behaviors so that the AI-based algorithm can distinguish it without a high misclassification rate. With DL, different models improved automatically the feature engineering process by also detecting unknown threats. Despite the progress of AI-based classification, adversarial attacks targeting AI systems have emerged [70], allowing certain malicious applications to evade detection and remain undetected after installation. In addition to adversarial evasion,

attackers have developed several *anti-analysis* mechanisms. These include *anti-static* techniques that hinder decompilation, a*nti-dynamic* mechanisms that detect emulated environments and block execution, and *obfuscation* strategies that intentionally disguise code semantics to prevent reverse engineering [168, 169]. Such countermeasures make malware analysis increasingly complex, forcing researchers to explore more resilient and adaptive detection pipelines.

It is important to note that these analysis methodologies are not limited to malware detection but are also crucial to vulnerability discovery. Vulnerability assessment is fundamental to reduce the attack likelihood, the damage in the infrastructure, prevent the threat. Most current malware and attacks exploit a vulnerability to enter in the system and perform their malicious activity (e.g. steal data, run unauthorized malicious code, encrypt files). An Android application is typically composed of Java code, which defines the main logic and user interface, and C/C++ native code, which manages performance-intensive or hardware-dependent functionalities. Both components can contain vulnerabilities, unintended flaws that may lead to insecure or unstable behavior [125, 219]. Vulnerabilities frequently arise from improper data management, unsafe user input, or misuse of system APIs. Common examples include buffer overflows, format string vulnerabilities, and use-after-free errors. These issues can result in memory corruption, application crashes, or arbitrary code execution. Critically, such weaknesses are often exploited by malware to gain unauthorized privileges, bypass sandboxing, or exfiltrate sensitive data, establishing a direct and practical link between vulnerability analysis and malware research [170].

Among the different vulnerability classes, those affecting the Android native layer (i.e. C/C++ code) are particularly critical. When exploited, they can grant direct access to the device's main memory (i.e. RAM), where sensitive data is often stored in clear text. This allows the attacker to perform silent and persistent attacks that remain active until the device is rebooted. The threat becomes especially severe in industrial and critical infrastructure contexts, where devices are rarely powered off.

Although the current literature offers several static and dynamic vulnerability-assessment methods, important limitations remain. Among all we can find APK with anti-analysis techniques sometimes even derived from unofficial repacked versions with malicious purposes; malware with evasive and adversarial behavior; emergent threats (e.g. fileless malware, stegomalware, wasm, obfuscation); insufficient robust detection mechanisms and the problem of feature engineering. According to this, the main question that lead to this thesis is ***Are features extracted from Digital Forensics analysis robust for malware detection, in particular memory forensics?***. To answer to this question, currently there is no public available dataset and methodologies, in particular in Android fundamental to protect mobile users. According to the main problem stated before, three main questions arise automatically:

1. How to access the content of the RAM?
2. How to acquire the RAM?
3. How to analyse the RAM?

In this thesis, I address the following directions. First of all, the RAM content can be accessed by exploiting native code vulnerabilities (C/C++ libraries) and reside in RAM. Hence, the main question of the first part is ***How to exploit Native Code vulnerabilities and reside in RAM?***

The other two questions can be better addresses with the following one: ***How to forensically acquire and analyse a non-rooted Android device?*** Immediately related to the main question of this thesis, derives the third research question: ***How to apply Artificial Intelligence to Digital Forensics?***. By following these research questions, the thesis is structured in four main Chapters: chapter 5 to analyse the native code vulnerabilities in order to understand how to access the main memory of an Android device; chapter 6 and chapter 7 to address how to acquire and analyse the main memory of an Android device and lastly, chapter 8 with the study on how to apply AI to DF.

In this thesis, I address the Native Code vulnerability part in three directions. First, I estimate a risk score for each APK based on known Common Vulnerabilities and Exposures (CVEs). Second, I reconstruct the call graph from Java to native code to assess the reachability of vulnerable functions. Third, I perform input-based vulnerability discovery through fuzzing. Graph Neural Networks (GNNs), a class of deep models designed for graph-structured data, can be used to identify vulnerable paths (i.e. code flows leading to vulnerable functions), reconstruct stripped binaries that have lost symbol information during compilation, and associate analyzed native code to specific libraries such as OpenCV, OpenSSL, or LibPNG[30]. Additionally, AI can improve fuzzing by prioritizing test cases, predicting execution coverage, and dynamically adapting inputs, while Large Language Models (LLM) can guide the fuzzing process by generating context-aware test inputs derived from reverse-engineering knowledge or binary similarity [217].

Due to the rise of stealthy threats such as malware employing anti-analysis, adversarial, or steganographic techniques, memory forensics has emerged as a powerful complementary defense. Memory forensics focuses on the extraction and analysis of volatile data from the main memory (RAM) of a device [53, 126]. Various approaches exist for Android memory acquisition, yet they face three persistent challenges: the need for super-user privileges (rooting), which the Operating System (OS) restricts; the requirement of forensic soundness to ensure data integrity and repeatability; and the absence of public datasets, which hinders reproducibility and the training of robust AI models [23].

To address these issues, this thesis proposes the use of a mobile Digital Twin (mDT) to perform forensic memory acquisition on Android systems without requiring the physical device to be rooted. The proposed framework called `MoLIFE` establishes a controlled and reproducible environment that respects the principles of data preservation in forensic contexts. Additionally, I provide a unified methodology for Android memory dumping (i.e. `AndroMemDump`) and apply it to the analysis of representative malware families, including stegomalware[69, 188, 192], droppers, trojans, and adversarial samples. Finally, I propose multimodal data-encoding strategies (i.e. image, audio, and string representations) for robust and scalable AI-based analysis.

AI-based techniques play a central role in this work because they provide a flexible, scalable, and interpretable toolbox for representing, detecting, and explaining malicious behavior across multiple data structures. When memory dumps or binaries are encoded as images (for example, by mapping byte values to pixels in 1D images or constructing 2D byte matrices), CNNs can exploit spatial and long-range patterns that correspond to the structural or behavioral signatures of malware. These visual encodings make it possible to spot recurring byte-level patterns, packed regions, or embedded payloads that are hard to capture with purely syntactic features [29, 68]. Audio encodings (i.e. treating raw byte streams as waveforms or converting sequences to spectrograms) allow ML algorithms

and waveform models, even based on CNNs, to capture temporal rhythms and frequency patterns that reflect execution traces or repeated code idioms. Such representations are particularly useful for surfacing periodic behaviors or repeated payload fragments that standard static analysis may miss. Textual/string encodings obtained from memory (e.g. extracted ASCII/UTF strings, decompiled code snippets, or disassembled opcode sequences) are naturally amenable to Transformer-style models and LLM, which can model syntax and semantics, detect anomalies in API usage, and infer likely intent even when obfuscation is present [79, 81, 159].

Beyond pure detection, these multimodal encodings facilitate richer forms of data explanation. Deep models do not merely output labels; through attention maps, saliency methods, class activation mapping, and gradient-based attribution, they can highlight which image regions, audio frames, or string tokens drove a decision, thereby producing human-interpretable evidence that can be inspected by an analyst and used in a forensic report. When a CNN flags a memory image as malicious, corresponding saliency maps can point to the byte ranges responsible for the classification; when an LLM flags suspicious string sequences, attention scores or token attributions can indicate which API calls or constants are anomalous. This traceability is essential for forensic admissibility and for analysts seeking to understand root causes rather than accept opaque labels.

Graph representations derived from code and runtime artifacts (e.g. control-flow graphs (CFGs), call graphs, and data-flow graphs) are another critical modality. GNNs operating on these graphs can learn structural patterns of vulnerability and exploitability, prioritizing likely vulnerable paths, predicting whether a native function is reachable from user-controlled inputs, and helping reconstruct stripped or obfuscated binaries by leveraging graph similarity to known libraries. Integrating CFG/call-graph analysis with learned embeddings enables the system to reason about reachability and exploitation in a manner that purely statistical encodings cannot, thereby bridging the gap between static program analysis and learned pattern recognition [98, 229].

Fuzzing benefits enormously from AI augmentation. Traditional fuzzers generate large volumes of inputs with limited guidance; AI-driven fuzzing can prioritize inputs that maximize code coverage or target vulnerable paths inferred by GNNs and CFG analysis. Reinforcement learning agents, coverage-guided neural generators, and LLMs can propose semantically meaningful inputs that traverse deeper program states, dramatically increasing the chance of triggering complex, context-sensitive bugs. LLMs, in particular, can synthesize inputs that conform to protocol grammars or application-specific formats by leveraging learned token distributions and prompt conditioning, thereby producing more effective test cases than blind mutation strategies [217].

LLM serve multiple roles in this pipeline beyond input generation. As powerful sequence models, LLMs can assist in data interpretation by summarizing disassembled snippets, explaining likely function purposes, and proposing hypotheses about exploit paths. Such analysis traditionally requires extensive manual reverse engineering. They can be used to generate natural-language explanations of model outputs, annotate suspicious code regions, and even draft forensic narratives grounded in the evidence produced by multimodal analyses. Moreover, LLM can be used to produce synthetic yet realistic benign and malicious samples for training and evaluation, thereby helping to address dataset scarcity while allowing for controlled experiments on adversarial robustness. When used for data generation, however, it is essential to ensure that no distributional biases or artificial artifacts are introduced, as these could distort the data characteristics and compromise the validity of the generated samples. These samples should be validated against real traces and, where possible,

labeled or filtered by domain-specific heuristics.

The combination of these AI capabilities yields a closed-loop system for detection, explanation, and proactive testing: image/audio/string encodings feed detectors that point analysts to suspicious memory regions; CFG and GNN analyses identify vulnerable or reachable functions; AI-guided fuzzers and LLMs generate targeted inputs to validate exploitability; and explainability techniques translate model decisions into human-readable evidence that supports forensic conclusions. This multimodal, AI-centric approach therefore not only improves detection accuracy and scalability, but also enhances interpretability and investigative features that are indispensable in forensic contexts and that substantially extend what is achievable with non-AI, rule-based techniques.

Beyond classification, AI models contribute to feature extraction, representation learning, and interpretability. Deep networks can automatically derive high-level features that correlate with malicious behaviors, drastically reducing the need for manual feature engineering. Representation learning allows for the embedding of complex data (e.g. binary structures, system calls, or opcode sequences) into lower-dimensional spaces where semantic similarities become measurable and actionable. This capacity to discover and exploit abstract relationships between features makes AI-based methods far more powerful than traditional static or signature-based analyses.

Furthermore, AI models can continuously adapt to new threats and evolving attack vectors. Through fine-tuning, correct feature selection and continual learning, models can update their internal representations as new malware families emerge, maintaining high detection accuracy over time [64]. This adaptability is particularly relevant in the Android ecosystem, where the diversity of applications and the rapid evolution of malicious samples challenge the scalability of conventional approaches.

Finally, AI enhances explainability and decision support. Techniques such as attention mechanisms, saliency mapping, and feature attribution provide insights into which portions of the data most influenced a classification or detection decision. Such interpretability is crucial in digital forensics and cybersecurity investigations, where analysts must justify their conclusions under established evidentiary standards. By combining automation, adaptability, and interpretability, AI-based systems offer a unified analytical framework capable of handling the scale, complexity, and variability that characterize modern Android malware and forensic data.

In the broader context of Digital Forensics (DF), this thesis also examines how AI can be integrated into each DF phase, from data acquisition to analysis and interpretation. At the same time, it evaluates the robustness of AI-based forensic tools when exposed to adversarial manipulations such as deepfakes, morphing attacks, and data poisoning. This reflection extends to the impact of generative AI on digital investigations, particularly in content authenticity verification and evidence validation [92].

The main objective of this research, represented in Figure 1.1 is thus to combine Android security, AI, and DF into a unified analytical framework that strengthens malware detection, vulnerability assessment, and memory analysis in a forensically sound manner. The remainder of this thesis is structured as follows. Chapter 3 introduces the background on Android architecture, malware analysis, and memory forensics. Chapter 4 discusses related work on Android security, AI-based detection, and anti-analysis techniques. Chapter 5 explains the vulnerabilities in the Android Native Code, starting from the vulnerability risk estimation, exploring the vulnerability reachability, and retrieving the exploitation input. Chapter 6 presents the memory acquisition methodology for An-

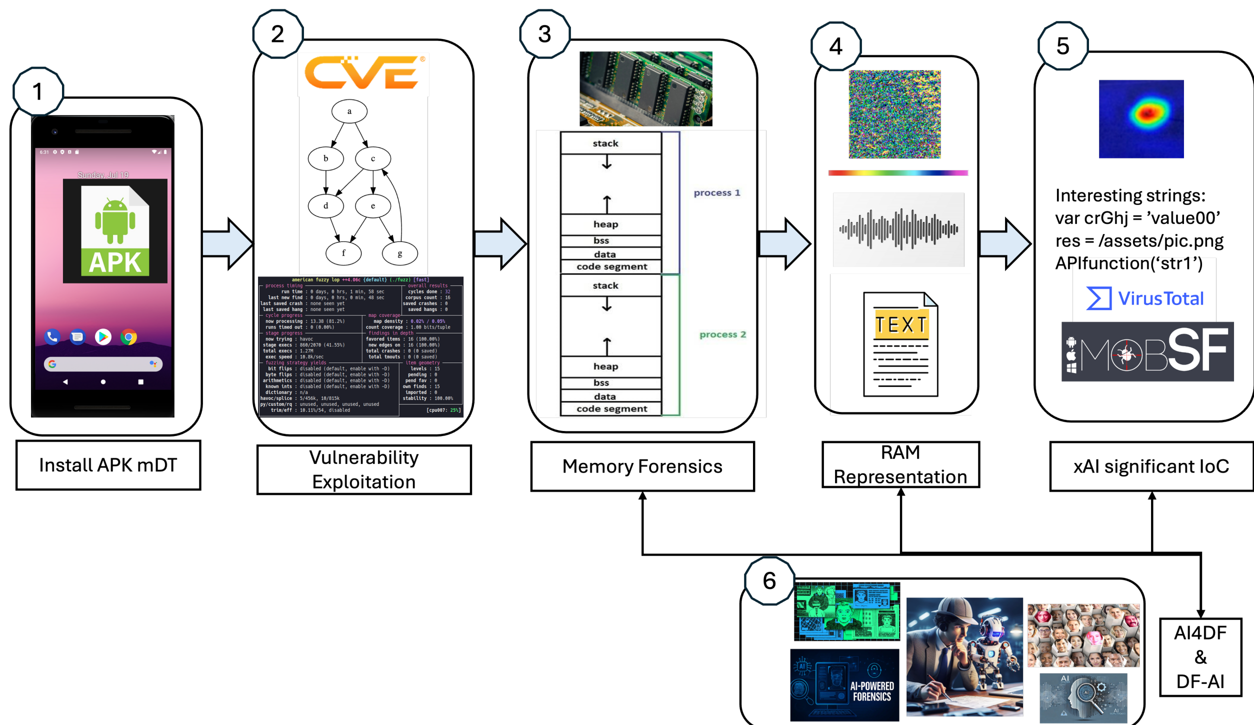

Figure 1.1: The Figure shows the full pipeline: block 1 is the APK installation; block 2 is the vulnerability discovery and exploitation; block 3 represents the Memory Forensics extraction, followed by RAM representation in block 4, from which the most significant IoC are extracted. These last phases are coordinated by the use of AI in DF

droid devices. First of all I describe the rooting problem in Android systems and a possible solution with Digital Twin; subsequently, the possible Memory Forensics methodologies for data extraction are depicted. Subsequently, Chapter 7 shows the different encodings I adopted for Android memory analysis to identify running malware (e.g. images, audio, text). Finally, Chapter 8 designs how AI can be applied to DF pipelines, the importance of its robustness, and the future DF investigations of generative-AI multimedia. Chapter 9 summarizes the main contributions and future directions of this work.

Chapter 2

Contributions of this thesis

In the following I will highlight the main contributions of this thesis in the field of Artificial Intelligence for Cybersecurity applied to Android devices. First of all, the main contribution is to study Android devices, the most used mobile OS worldwide, focusing on stealthy attacks (i.e. malware employing obfuscation, encryption, steganography, native code vulnerabilities, anti-analysis and adversarial behavior), analysing and detecting the threat with Memory Forensics techniques. In fact, the main objective of this thesis is the union of DF to detect cyberattacks in Android devices, converting the DF analysis into features for AI-based algorithms (e.g. CNN, NLP, LLM, traditional ML). Additionally, it points out some guidelines on the general application of AI into DF analysis, the future of DF investigations with the emergent multimedia files made by Generative Artificial Intelligence (genAI), studying the robustness of actual AI-based DF tools.

The main methodological contributions are based on three main pillars: *(i)* **Vulnerability Assessment** to detect and exploit vulnerabilities in the Native Code of Android applications; *(ii)* **Forensics Acquisition** to acquire forensically sound data in the Android device; and *(iii)* **AI-multimodal detection** by encoding the acquired forensics data for a robust detection.

In the **Vulnerability Assessment** field (chapter 5), I propose an integrative workflow to analyse the Android Native Code (compiled C/C++ libraries). The framework looks for published CVE (vulnerabilities) in the decompiled native library and assigns a risk score to prioritize its assessment. Subsequently, with the use of graphs it is possible to find all possible paths and the shortest one to call the vulnerable function, i.e. reach the vulnerability. The third and last step is the exploitation of the vulnerability, where AI-based fuzzing techniques (i.e. input generated with LLMs) are applied to find the input triggering the vulnerability. If successfull, the attack resides in RAM where all data are in cleartext, sensitive information could be read, specific commands and stealthy attacks could be executed to not be detected.

At this point comes the **Forensics Acquisition** part (chapter 6), where I developed the MoLIFE framework to analyse and detect attacks in Android devices, having a complete overview before (i.e. at the installation time), during (i.e. at execution) and after (i.e. reconstruct the incident) the threat, proposing a solution for the rooting problem in DF acquisition. In fact, a complete DF acquisition, including the main memory is possible only if the Android device is root, i.e. owns super user privileges due to OS and kernel protections. Because of this, in traditional DF investigations when involving Android devices, if the device is not rooted before the analysis, it is not acquired completely. With `MoLIFE`, I propose the application of DF to mobile devices, hence the mDT in

order to have three copies (before, during and after) to analyse the threat and attack. During tests, we discovered that the mDT can be applied even only after the incident for the complete acquisition of mobile data. Additionally, I present `AndroMemDump`, a tool for the complete acquisition of Android main memory, divided into the full RAM acquisition or the target-APK memory.

After the acquisition, data must be analysed and threat detected automatically. In this part (chapter 7), I propose an AI multimodal encoding and detection where the acquired data can be analysed as it is by a human analyst or converted into images, audio or text for automatic detection with Convolutional Neural Network (CNN), LLM, Natural Language Processing (NLP) algorithms. After the detection, we add a layer of Explainable AI (xAI) to explain the automatic classification and retrieve the most important bytes that derived such detection. In this way, the human analyst can better understand why the algorithm took that specific classification and analyse deeply the original forensics data. Generally, xAI is very important in AI-based DF analysis. Currently, many commercial tools use AI algorithms to detect and classify data inside DF evidence. In the last part (chapter 8), I show the robustness of such tools with two different dataset of experimental setup. Additionally, I highlight how AI can be used in the full DF investigation and the future of DF analysis with the emergent genAI multimedia files.

The thesis has been written on the following published papers [174–176, 178, 179, 188]. Some of the presented works are not still published but under review into important journals and conferences. Additionally, I worked on other papers not strictly related to this thesis: [155, 180, 189].

Chapter 3

Technical Overview of Android System and Digital Forensics

This Chapter introduces all the relevant concepts in order to understand this work. The first section is focused on Android and the second one on Digital Forensics.

3.1 Android System

Android is the OS for mobile devices most used worldwide [4]. This is because it is widely adopted by different productive houses (e.g. Samsung, Oppo, Xiaomi, etc.), it is open source and can be easily adapted to different use cases (e.g. personal use, working assistant, monitoring system, etc) and Android devices are cheaper than iOS ones. Android is not used only for smartphones and tablets but also smartwatches, smart TVs, and automotive. As a consequence, Android applications are the most downloaded and developed applications worldwide. Originally developed by Android Inc. in 2003 and later acquired by Google in 2005, Android was designed as a flexible and open platform for mobile devices. The first commercial version, Android 1.0, was released in 2008 alongside the HTC Dream (T-Mobile G1), marking the beginning of the Android ecosystem. Over the years, Android has evolved through numerous versions, each improving security, performance, and user experience, while maintaining backward compatibility through its layered architecture based on the Linux kernel. Today, Android is maintained under the Android Open Source Project (AOSP), led by Google, with continuous development of new features, system services, and developer tools.

This section will first explain the structure of the Android system with an in-depth analysis of the memory structure. Subsequently, a description of the Android application with details on the use on Native Code.

3.1.1 Android OS

The Android OS is organized into five main layers, as shown in Figure 3.1, where at the top is the application layer, where all user-installed and system apps reside. This layer allows developers to extend the functionality of the OS without altering deeper system components. Beneath it lies the Android framework, which provides a vast set of APIs that enable applications to interact with device resources such as user interface elements, data storage, and inter-component communication.

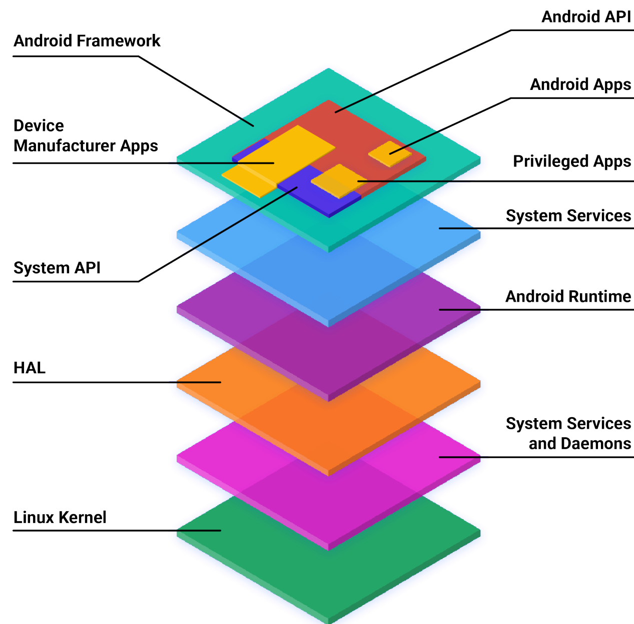

Figure 3.1: The picture shows the level of abstraction in the Android OS. Image taken from the official Google repository

The framework acts as the intermediator between applications and the runtime environment, incorporating non-app-specific code that runs inside the Dalvik Virtual Machine (DVM). The next layer is composed of native libraries and core system services, much of which are derived from open source Unix projects. These libraries implement low-level functionalities of the framework, while the system services handle device-specific initialization, hardware management, and security-critical operations such as debugging. Below there is the Hardware Abstraction Layer (HAL) and its Hardware Interface Description Language (HIDL), which define standardized interfaces between the software and hardware drivers. This design allows the OS to remain hardware-agnostic while enabling driver updates without disrupting the OS. At the base is the Android-modified Linux kernel, which differs from the standard Linux kernel in many ways, including customizations in file systems, networking, and memory management. It is responsible for managing physical resources, process communication, app certificates, file encryption, password and biometric authentication, and memory protection.

Android enforces security through two complementary permission models. At the low level, the Linux kernel applies a sandbox model that controls access to files and system resources by assigning each app a unique user ID and group permissions. At the high level, the Android runtime enforces permissions declared in an app's manifest, mapping these to the underlying OS access controls during installation. This dual approach provides both granular resource protection and user-facing permission management.

Originally, Android applications were executed on the DVM, a register-based Virtual Machine (VM) designed for efficiency in low-memory environments. DVM compiles Java-like source code into .class files, which are then transformed into Dalvik bytecode and packaged into .dex (Dalvik Executable) files. These are optimized before being interpreted by the VM. Starting with Android 5.0 (Lollipop), DVM was replaced by Android Runtime (ART), which compiles bytecode into native machine instructions (Ahead-of-Time (AOT)) during installation. This change greatly improved

performance, reduced runtime overhead, and enhanced battery efficiency.

A critical component in Android's startup and app execution process is the Zygote process. Launched early during system boot, Zygote preloads core libraries and system classes into memory. When a new application needs to start, Zygote forks itself, creating a new process that inherits these preloaded components. This design eliminates the need to repeatedly load the framework for each application, significantly reducing app startup times and preserving system resource consumption.

Beyond its layered architecture, Android relies on a strict privilege hierarchy derived from Linux's user-based access control model. The most privileged account in this hierarchy is the super-user, commonly referred to as root. By default, root privileges are disabled to preserve device integrity and prevent unauthorized access to system partitions or security-sensitive operations. Rooting is the process of obtaining super-user privileges, effectively granting unrestricted control over the operating system. From a security standpoint, this operation bypasses Android's sandboxing and permission enforcement, allowing direct access to protected directories such as /data, where applications and user artifacts are stored. The rooting process generally begins with bootloader unlocking, since the bootloader is locked by default to block unverified firmware. The bootloader is responsible for initializing hardware and loading the OS. When permitted by the manufacturer, the bootloader can be unlocked using the oem unlock command from recovery mode, a dedicated partition for maintenance operations. Once unlocked, a patched boot image or custom ROM can be flashed to enable root privileges. Popular tools for this process include Odin, commonly used for Samsung devices, and Magisk, which modifies the stock boot image to inject super-user capabilities while preserving system integrity. In cases where official unlocking is not possible due to vendor restrictions, researchers and advanced users may resort to privilege-escalation exploits targeting vulnerabilities in the kernel or firmware. Root access can be verified using the Android Debug Bridge (ADB) by opening a shell, executing the su command, and confirming the user identity with whoami. A successful rooting operation will return root, indicating full administrative privileges. However, rooting carries risks: it can modify the device's state, trigger a factory reset, or invalidate digital evidence, posing challenges for forensic soundness. Rooting an Android device is often necessary in DF because it grants super-user privileges, allowing access to protected system areas (such as /data) and enabling complete data acquisition (including filesystem, physical, and volatile (RAM) extractions) that are otherwise inaccessible on unrooted devices. In essence, rooting is required to recover deleted, encrypted, or hidden artifacts and to perform low-level forensic operations, though it must be done cautiously and lawfully to avoid altering evidence or compromising its admissibility. From a cybersecurity perspective, rooting is generally discouraged because it bypasses Android's built-in security model and exposes the device to significant risks. By granting super-user privileges, rooting disables critical protections such as sandboxing, verified boot, and system integrity checks, allowing malicious apps to access or modify system files, escalate privileges, or steal sensitive data. It also invalidates security updates and warranties, increases the risk of persistent malware and data sensitive exfiltration. In short, while rooting enables deeper control, it weakens the device's overall security and opens new attack surfaces. For this reason, it is preferred to not apply it in critical infrastructures.

Android Emulation

This paragraph is fundamental to better understand some constraints and limitations of the developed works, in particular dealing with the vulnerability exploitation (Table 5.3) and DF acquisition of the emulator (Figure 6.1) where the knowledge of the Android emulation is essential. An Android emulator is, at its core, a virtual environment that simulates an Android device so that the OS and apps behave as if they're running on real hardware. In practice, this usually relies on a VM manager such as QEMU, VirtualBox, or a custom virtualization layer. These tools simulate key components like the CPU, RAM, GPU, and input/output devices. The difference between simulation and virtualization is important: virtualization runs code directly on your real CPU if the instruction set matches, while simulation translates instructions from one architecture to another, which is slower but more flexible.

In Android Studio [32] built-in emulator, Google provides prebuilt system images, which are essentially Android OS snapshots compiled for specific architectures like ARM, x86, or x86_64. If the emulator is running on a PC with an Intel or AMD processor, using an x86 system image means the emulator can execute code almost directly with the help of hardware virtualization extensions like Intel VT-x or AMD-V. This is much faster than emulating an ARM image on an x86 processor, which requires translating every CPU instruction in real time. Full emulation, where the guest CPU architecture is ARM and every instruction is faithfully interpreted, would be ideal for exact hardware behavior, but it is significantly slower unless the host CPU is itself ARM-based. That is why many developers now use x86-based Android images for speed during development, even though real phones mostly use ARM chips.

Genymotion [88] operates on similar principles but wraps its virtual Android devices in a polished interface and often runs them inside VirtualBox or the cloud. It offers features like quick switching between different device profiles, network throttling, and GPS mocking, making it attractive for testing a wide variety of scenarios without physically owning dozens of devices.

The difference between an emulator and a real Android device goes beyond CPU architecture. Physical devices have sensors such as accelerometers, gyroscopes, ambient light detectors, barometers, and hardware-specific radios for Wi-Fi, LTE, Bluetooth, and NFC. While an emulator can simulate sensor data, it cannot perfectly mimic the electrical quirks, latency, signal interference, or manufacturer-specific firmware behavior. Graphics performance can also differ because in an emulator, the GPU may be partially virtualized or mapped through the host's OpenGL drivers, which is not always identical to the real mobile GPU pipeline. Battery usage patterns, thermals, camera processing, and proprietary driver optimizations are other areas where the gap is visible. This is why experienced developers use emulators for rapid iteration and debugging, but still run final tests on actual hardware to catch edge cases that software simulation might miss. This topic will be addressed in Chapter 6.1.

Android Memory Management

Android devices use LPDDR (Low Power DDR) DRAM as their physical RAM, which is volatile memory divided into small units called pages, usually 4 KB each. This RAM doesn't belong entirely to apps, but is orchestrated by the Linux kernel that Android runs on. The kernel manages memory by giving each process its own virtual memory map, translating virtual addresses into physical ones

through the CPU's Memory Management Unit. It also decides which pages stay in RAM, which get compressed into zRAM swap space, and which processes to kill if the system runs low on memory.

When an Android device is powered on, the bootloader loads the Linux kernel into RAM. The kernel sets up drivers, allocates space for itself, and starts the init process, which then launches system daemons. A system daemon is a background process automatically launched, running continuously without user interaction to manage essential OS services such as networking, power management, storage, and logging, ensuring the stable and coordinated functioning of all system components. An example of daemon is Zygote, a special process that preloads Android's core classes and resources into memory. Zygote's job is to fork new processes whenever you open an app. Because of Linux's copy-on-write mechanism, all apps forked from Zygote share the same preloaded framework code and only get their own separate memory when they modify something.

When the user taps an app icon, the launcher communicates with the ActivityManagerService, which instructs Zygote to fork a new process. This process inherits the shared Android framework code from Zygote, then loads the APK 's DEX bytecode and resources. Much of this is memory-mapped from storage rather than fully copied into RAM, which saves space and speeds up loading. Native libraries inside the APK are also memory-mapped, and any compiled code produced by ART Just-in-Time (JIT) or AOT compilation live in the app's own address space alongside its Java/Kotlin heap.

RAM in Android is never intentionally kept empty. Background apps and cached processes are left in memory so they can reopen instantly. If RAM pressure builds, the kernel first reclaims disk caches, then kills the least recently used background processes, keeping the foreground app safe. Shared memory mechanisms like ashmem, graphics buffer sharing through ION or DMA-BUF, and binder-based IPC allow different processes to communicate and exchange data without making extra copies in RAM. From boot time to running an app, Android's RAM is a constantly shifting map of kernel space, shared system code, app-specific heaps, and cached data. The kernel and Android framework work together to reuse memory wherever possible, keeping the system responsive while balancing the needs of multiple apps and services.

3.1.2 Android Application Package

An APK [5] shown in Figure 3.2, typically written in Java or Kotlin, and designed to run on Android-powered devices such as smartphones, tablets, and smart TVs. The Android OS is built on the Linux kernel, and native code written in C or C++ is often integrated to interact directly with core system functionalities and hardware components such as the camera and microphone. Each application is packaged into an Android Package file, with the extension .apk, which is the product of the compilation process carried out through Android Studio or other compatible development environments (e.g. VSCode or Xamarin). While Java remains the predominant development language, Kotlin has become increasingly popular. Java is a stable, verbose, and widely supported language suited for legacy or cross-platform Java Virtual Machine (JVM) projects, while Kotlin is a modern, concise, and safer alternative preferred for new Android development due to its built-in null safety, coroutines, and improved readability. C/C++ is used in projects that require native performance or hardware-level access.

The .apk file contains all resources required for the application's installation and execution. Its structure includes compiled native libraries stored in the lib directory, organized for each CPU

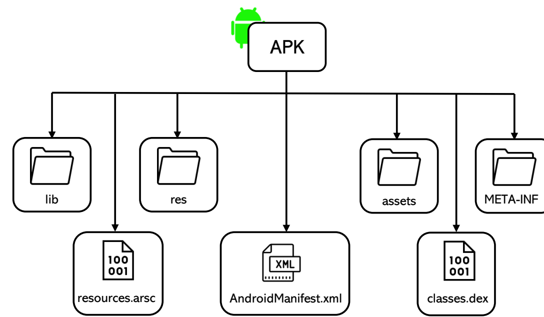

Figure 3.2: The picture shows the main directories found in an APK

architecture (e.g. ARM, ARM64, x86, and x86_64); Dalvik Executable (.dex) files containing the compiled classes and bytecode to be executed by the ART; the AndroidManifest.xml file, which defines the app's permissions, entry points, components, required hardware and software features, API compatibility, and intent filters for inter-application communication; the META-INF directory containing the application's cryptographic signatures, manifest information, and metadata necessary for verifying its integrity; the res directory holding compiled resources such as XML layouts, animations, styles, color definitions, menus, and device-configuration-specific assets; and the assets directory, which stores uncompiled resources such as images, audio, or raw files that are accessed programmatically. Precompiled resources, including XML files and binary data, are stored in the resources.arsc file, where each resource is assigned an integer identifier for runtime retrieval through the R class. During the build process, the Java (or Kotlin) source code written in the project is first compiled into Java bytecode (.class files) using the Java compiler. These class files are then processed by the Android build tools, which convert them into Dalvik Executable (.dex) files optimized for the ART. The intermediate representation of this bytecode is written in Smali, an assembly-like language used to describe the instructions contained within .dex files. Each .smali file corresponds to a Java class and exposes low-level operations, registers, and method calls exactly as executed by the Dalvik or ART virtual machine. Smali is particularly useful for reverse engineering or manual modification of Android applications, as it allows analysts to inspect or edit the logic of compiled code without access to the original Java sources. At the same time, resources are compiled and linked, and the Android Asset Packaging Tool (AAPT or AAPT2) packages them together with the manifest and compiled code into a single APK file, which is then digitally signed and aligned (zipalign) to produce the final installable application.

From a system perspective, every Android application operates within its own security sandbox, running in a dedicated Linux process with a unique user ID (UID) assigned by the OS. This UID isolates the application's data from that of other apps unless multiple applications are deliberately configured to share the same UID, in which case they can access shared files, run in the same process, and share the same virtual machine instance (e.g. two companion apps from the same developer that declare the same sharedUserId (such as "com.example.shared") and are signed with the same certificate to enable direct data and resource sharing). Up to Android 4.4 KitKat, applications were executed by the DVM, a register-based VM optimized for mobile devices. Dalvik differed from the JVM in its bytecode format, exception handling, and register-based architecture, and employed JIT compilation, compiling portions of the code during execution. From Android 5.0 Lollipop onward,

DVM was replaced by the ART, which uses AOT compilation at installation, improving runtime efficiency and reducing memory overhead.

When launched, an Android application is forked from the Zygote process, which preloads core classes, system libraries, and essential resources into memory to optimize startup time and resource usage. ART then loads the application's .dex files, executing the code and applying optimizations where applicable. Dynamically loaded Java or Kotlin objects are managed by the garbage collector and stored in the Dalvik/ART heap, while certain runtime data may reside in anonymous memory mappings with no corresponding file on disk. The device's RAM also contains the stack section for each active thread, storing function call frames, local variables, and execution context. These local variables can include strings representing API names or Java/Kotlin method names, which are particularly relevant for the use of reflection, i.e. a language feature that allows inspection and invocation of classes, methods, and fields at runtime without knowing their names at compile time. In the case of stegomalware [69, 188, 189], reflection can be exploited to dynamically load and execute a hidden payload stored as a string.

The Android framework organizes application behavior into four main component types: *activities*, which serve as entry points for user interaction and represent individual screens or actions; *services*, which handle background processing either until a task is completed (started services) or while serving other applications (bound services); *broadcast receivers*, which listen for and respond to system-wide or application-specific broadcast messages; and *content providers*, which manage structured data storage and facilitate secure data sharing between applications. These components operate asynchronously and together define the functional behavior of the app.

From a resource management perspective, Android selects and loads resources at runtime based on the device configuration, such as screen size or density, using identifiers from the R class. The res directory is organized into subdirectories with descriptive names according to content type and purpose, such as animator, color, font, layout, xml, raw (for audio files), drawable, and mipmap (for launcher icons of different densities). Assets remain in their original format and can be accessed programmatically without compilation.

The internal structure and execution process of Android applications are particularly relevant when considering malicious repackaging. An attacker aiming to embed a payload into an existing application must first decompile the .apk using reverse-engineering tools such as ApkTool [212] or JADX [184]. ApkTool allows decoding of resources to nearly original form and disassembly of Dalvik bytecode into human-readable smali files. Once decompiled, the attacker can inject new code, modify existing classes, replace or add native libraries, or alter resources such as images, audio, and XML layouts. In some cases, permissions must be updated in the AndroidManifest.xml to enable the new functionality. After modifications, the .apk is rebuilt, and because the legitimate developer's signing key is unavailable, it must be re-signed with a new certificate using tools such as apksigner.

This repackaging process generally follows four stages: *(i)* decompiling the application to extract code and resources; *(ii)* modifying files as required; *(iii)* rebuilding the application; and *(iv)* re-signing the package. The resulting repacked application will differ from the original in its signature, potentially in its resource structure, and in its executable code. However, the process may fail if the target application employs anti-tampering techniques designed to trigger errors during recompilation or runtime execution when unexpected changes are detected.

Overall, understanding the composition of an Android application (from its packaged structure and resource management to its runtime behavior and potential for modification) is essential both for legitimate development and for analyzing the risks posed by malicious repackaging, particularly in the context of embedding concealed payloads or implementing advanced obfuscation techniques such as reflection-based execution.

APK Native Code

In the Android OS and APK, one of the important layers is the Native Library layer, which plays a key role in enabling developers to reuse existing code written in C/C++. Rather than having to reimplement certain features from scratch, developers can integrate pre-built libraries that already contain the required functionality. This not only speeds up the development process, but also ensures that code is portable across different platforms, Android system versions, and CPU architectures. For example, each application may have separate native library versions to match the ARM or x86 architecture of the target device. The use of native libraries also typically improves application performance. To take advantage of these capabilities, developers rely on the Native Development Kit (NDK) [3], an Android Studio tool designed for importing and managing native libraries. The NDK makes it possible to run computationally intensive tasks (e.g. those in games or physics simulations) with lower latency, sometimes even offloading heavy processing from the device its Executable and Linkable format (ELF). It allows developers to compile C/C++ code into native shared libraries, which are stored in the lib directory of the APK and use the .so extension. The NDK provides two main approaches for implementing native libraries. The first uses native_activity.h, which defines the NativeActivity class along with its callback interfaces and data structures. This method runs entirely on a single thread, so callbacks must never be blocking. The second approach uses android_native_app_glue.h, a static support library that works on top of the native_activity.h interface. This creates a separate thread to handle callbacks and inputs, preventing the main thread from being blocked.

When building native code, Android uses an Application Binary Interface (ABI) to determine which library should be executed on a given CPU and instruction set combination. The ABI specifies details such as the CPU type, endianness (always little-endian in Android), the conventions for data exchange between the OS and applications, stack usage during function calls, and the binary format (always ELF on Android). Supported ABI include armeabi-v7a for ARM 32-bit processors with hardware floating-point support and Thumb-2 instructions; arm64-v8a for 64-bit ARMv8 CPUs with NEON registers and Android-specific register conventions; x86 for IA-32 architecture; and x86_64 for 64-bit x86 processors, commonly used in Android emulators.

Native libraries in Android come in two main forms: shared libraries (.so), which are dynamically linked at runtime; and static libraries (.a), which are linked into other binaries at compile time. Developing an application that includes native code generally follows a sequence of steps: deciding which parts of the app will be implemented in Java and which in native code, setting up the Android project, describing the native library in an Android.mk file inside the Java Native Interface (JNI) directory, optionally configuring ABI and build options in Application.mk, compiling the native code with NDK-build, and finally packaging the application. The resulting native libraries are placed in the lib directory of the APK, in subdirectories named after their respective ABI.

Since native libraries are ELF binaries, analyzing them is essentially the same as analyzing any

other ELF file. An ELF file begins with a header that contains information such as a magic number to identify the file type, the file's architecture (32-bit or 64-bit), endianness, ELF version, ABI type, and entry point address. Following the header, the Program Header specifies how the OS should create a process image, and the Section Header lists all the sections in the file. The most significant sections are .text (containing code and read-only data), .data (for initialized static data), and .bss (for uninitialized static data). A variety of tools exist for analyzing ELF files, including readELF [], objdump[], and the Python library[], as well as more comprehensive reverse engineering platforms such as Ghidra[]. Developed by the NSA, Ghidra can disassemble and decompile binaries for multiple architectures, including ARM, MIPS, and x86, and offers both a graphical interface and a headless command-line mode. It supports script development in Python or Java, allowing automated extraction and processing of functions, symbols, and other program structures. Although it does not provide debugging features, Ghidra is a powerful tool for examining native libraries, inspecting call graphs, symbol tables, and memory maps, and generating detailed analyses.

Bridging the gap between Java and native code is the JNI allowing Java or Kotlin code to invoke functions written in C/C++, enabling developers to integrate high-performance native routines or use features not directly supported by the standard Java class libraries. Through JNI, it is possible to create, inspect, and modify Java objects, call Java methods from native code, handle exceptions, load classes dynamically, and even embed a JVM into an application using the Invocation API. The interface is built around two main data structures: JVM, which contains functions for managing the JVM its ELF, and JNI environmen. Over time, JNI has evolved from early, less portable implementations such as JDK 1.0's direct memory structure mapping, Netscape's JRI, and Microsoft's JVM interfaces, to Oracle's standardized version that unifies these approaches. When implementing custom JNI methods, developers are advised to minimize overhead by reducing data sorting, avoid asynchronous communication between Java and native code, limit thread creation, and keep the number of entry points between the two codebases small.

3.2 Digital Forensics

DF is a set of techniques to analyse digital devices in order to retrieve data. During the years different techniques have been developed and they evolve according to technology, developing new ones to new devices but also updating the old ones according to security patches. DF traces its origins to the late 1970s, when the growing presence of personal computers created an urgent need for systematic methods to investigate traces left behind by cyber incidents. In its early days, the field lacked standardized protocols or specialized methodologies, and investigative work was performed in an ad-hoc manner. The 1980s marked the beginning of formalized practices, with the creation of specialized investigative programs and units such as the FBI's Magnetic Media Program and the Computer Analysis and Response Team (CART) [54, 147, 158]. These initiatives began to shape early guidelines for evidence handling and analysis. By the late 1990s and early 2000s, the rapid increase in computer evidence drove the creation of international bodies like the International Organization on Computer Evidence (IOCE) and the first generation of standardized procedures, which emphasized repeatability, verifiability, and adherence to the chain of custody. A significant contribution to defining the discipline came from the Digital Forensic Research Workshop (DFRWS)[156], whose technical committee characterized DF as the use of scientifically derived and proven methods for the preservation, collection, validation, identification, analysis, interpretation, documentation, and presentation of evidence obtained from digital sources. The purpose is to reconstruct events found to be criminal or to anticipate unauthorized actions that could disrupt planned operations. This definition reinforced the notion that DF is a science, governed by rigorous methodology, evidential integrity, and legal admissibility. Generally, DF is divided into *live forensics* (i.e. the device is turned on, containing volatile important data that can be deleted as time passes, hence the acquisition is non-repeatable and sometimes is done with the collaboration of the device owner by giving passwords and PINs); and *post-mortem* forensics (i.e. the device is turned off and interesting data is stored permanently, hence the acquisition is repeatable and the forensic image is created).

DF operates according to a series of methodological and legal principles that ensure the reliability, reproducibility, and admissibility of digital evidence. Central among these are *integrity* (i.e. the acquired data must remain unaltered throughout the investigation and be verifiable through cryptographic hashes); *authenticity* (i.e. guarantees the evidence can be traced unambiguously to its source); *reproducibility* (i.e. ensuring that another expert following the same procedures would reach equivalent results); and *chain of custody* (i.e. a continuous documentation process recording who collected, handled, transferred, or analyzed each piece of evidence, under what conditions, and at what time). Other guiding principles include *proportionality* and *minimization*, which require investigators to limit their actions to the data strictly necessary for the case, and legal compliance, ensuring that evidence is acquired under the constraints of applicable laws and privacy regulations.

DF is generally structured into six main phases: *identification*, *preservation*, *acquisition*, *analysis*, *documentation*, and *presentation*. According to international standards such as those issued by the National Institute of Standards and Technology (NIST) and the International Organization for Standardization (ISO), this sequence is further detailed into subphases to ensure methodological rigor, reproducibility, and evidential reliability. Frameworks like NIST SP 800-86 [111] and the ISO/IEC 27037–27043 series [9, 14–16] have provided standardized methodological foundations

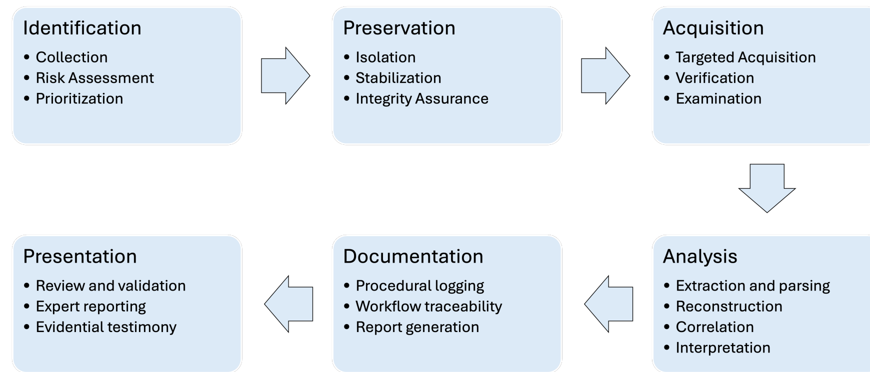

Figure 3.3: The picture shows the six main DF phases and their subphases, depicted in the order to be followed

for digital forensic investigations. series define a comprehensive and iterative forensic lifecycle, allowing investigators to revisit earlier steps as new findings emerge. For instance, the initial stages may include defining the investigation scope, identifying potential evidence sources, obtaining legal authorization, and establishing a controlled environment. Subsequent activities focus on preserving and acquiring data through isolation, imaging, and verification procedures, followed by systematic analysis, artifact correlation, and hypothesis validation. The process concludes with meticulous documentation and the formal presentation of results in a transparent, technically verifiable, and legally defensible manner.

In the following, each phase is described in detail. *Identification* represents the preliminary and strategic stage in which the investigator delineates the potential sources of digital evidence and defines the scope and objectives of the inquiry. This phase generally encompasses three sub-stages: *(i) collection*, consisting of the systematic enumeration of devices, network segments, or virtualized environments likely to contain relevant data, starting the chain of custody and taking pictures of the found evidence state; *(ii) risk assessment*, involving the evaluation of volatility, encryption status, and potential data loss due to ongoing system activity, hence the decision for live forensics; and *(iii) prioritization and planning*, which establishes the order of acquisition and the methodological framework to be adopted according to evidential relevance and resource constraints.

Preservation follows as a set of procedures designed to maintain the integrity and authenticity of digital evidence prior to and during acquisition. Its sub-phases typically include *(i) isolation*, which entails disconnecting or containing the target system or network to prevent further alteration or remote tampering; *(ii) stabilization*, where measures such as live memory capture or system suspension are employed to mitigate volatility while minimizing forensic contamination; and *(iii) integrity assurance*, realized through the computation of cryptographic hash values, secure storage of original media, and the establishment of a verifiable chain of custody that records every transfer, access, and procedural action.

Acquisition constitutes the empirical core of the forensic process, focused on the reproducible collection of digital artifacts in a manner that preserves evidential fidelity. This phase is articulated into *(i) physical acquisition*, involving the bit-by-bit duplication of entire storage devices or memory images using tools such as `dd`, FTK Imager, or LiME; *(ii) logical acquisition*, targeting specific file

systems, databases, or application data where full imaging is impractical; *(iii) selective or targeted acquisition*, used when investigative priorities or legal constraints restrict the scope of collection; *(iv) verification*, where the integrity of acquired data is confirmed through hash comparison between the original and its forensic replica; and *(v) examination* of the acquired extracted data.

Analysis is the interpretative phase in which the acquired data are transformed into meaningful evidential constructs. This stage generally comprises *(i) extraction and parsing*, wherein raw binary data are converted into structured artifacts through specialized parsers and forensic frameworks; *(ii) reconstruction*, aimed at reassembling file systems, event timelines, communication flows, or process hierarchies; *(iii) correlation*, involving the cross-referencing of heterogeneous data sources (e.g. disk images, memory dumps, and network captures) to establish causal and temporal relationships; and *(iv) interpretation*, which contextualizes the recovered evidence within the operational or legal narrative under examination, guided by both technical inference and domain expertise.

Documentation underpins the entire forensic workflow, maintaining transparency, reproducibility, and evidential accountability. Its internal sub-phases include *(i) procedural logging*, recording the temporal sequence of actions, parameters, and operator identities; *(ii) workflow traceability*, ensuring that every step of the investigation can be independently reconstructed and validated; and *(iii) report generation*, the systematic compilation of findings into structured, technically rigorous, and legally coherent outputs that clearly associate analytical results with their corresponding evidentiary sources.

Finally, *Presentation* constitutes the formal communication of results to judicial, corporate, or administrative stakeholders. This phase involves *(i) review and validation*, which subjects the findings to peer examination or automated verification to confirm their consistency and reliability; *(ii) expert reporting*, translating complex forensic evidence into an accessible yet scientifically grounded exposition suitable for non-technical audiences; and *(iii) evidential testimony*, where the forensic expert articulates and defends the applied methodologies, analytical reasoning, and derived conclusions in compliance with evidentiary standards and procedural law.

Together, these phases and sub-phases form a comprehensive procedural architecture that guarantees the methodological soundness of DF and ensures that its results are reproducible, transparent, and admissible within judicial and investigative contexts.

With the evolution of technology, however, DF has expanded its scope to address new domains such as mobile, cloud, and IoT forensics, as well as live memory acquisition, network traffic analysis, and open-source intelligence (OSINT). The increasing adoption of big data analytics and AI has further transformed the discipline, enabling investigators to process massive datasets, detect anomalies, and correlate events with unprecedented efficiency. Modern forensic practice also extends to emerging technologies, including autonomous vehicles, wearable health devices, blockchain systems, and industrial control infrastructures, where digital traces provide crucial insights for both criminal and civil investigations.

Over time, DF has come to encompass a variety of investigative approaches, each suited to different targets and evidence sources. Memory forensics [138] focuses on the volatile memory of devices such as computers and smartphones, revealing details about running processes, clear text data, encryption keys, and other ephemeral data that would disappear once a system is turned off. Disk or storage forensics [50] is concerned with persistent media like hard drives, SSDs, and removable storage, where deleted files can be recovered, hidden partitions revealed, and file system

activity reconstructed. Mobile forensics [103] addresses the acquisition and analysis of data from smartphones, tablets, and wearable devices, retrieving communications, app data, system logs, GPS records, and other usage artifacts. Network forensics [164] examines data flows across wired and wireless networks, enabling investigators to reconstruct intrusion attempts, track data exfiltration, and correlate communications with malicious activity. Cloud forensics [165] focuses on evidence stored within virtualized, distributed environments, often requiring specialized methods to overcome challenges like multi-tenancy, jurisdictional restrictions, and ephemeral data storage. Additional specialized areas have also emerged, including IoT device forensics [152], cryptocurrency transaction analysis [66], and multimedia forensics [72], each responding to the unique data structures and investigative requirements of their respective domains.

Central to all of these branches is the use of specialized tools designed to acquire, preserve, analyze, and present digital evidence without compromising its integrity. Acquisition tools vary according to the nature of the device and the type of data being captured. For instance, `FTK Imager` [21] is widely used in Windows environments to create bit-by-bit images of storage devices, while the Linux `dd` command is frequently employed for low-level disk acquisition on Linux-based systems, including Android devices when rooted (i.e. super user, admin access). Mobile device acquisition can be performed using commercial platforms such as Cellebrite's `UFED` [59], which can extract both physical and logical data from a wide range of mobile hardware. When dealing with volatile memory, investigators rely on tools like `FTK Imager`, `LiME` (Linux Memory Extractor) [19] to capture the full contents of RAM, or `fridump`[148] and `procmon` for extracting memory associated with specific processes. Once acquired, memory analysis is often performed using frameworks such as `Volatility` or its derivative, `Rekall` [163], which require an operating system profile to interpret the captured data correctly.

For analysis, investigators turn to comprehensive forensic suites that can process large volumes of data while preserving evidential integrity. `Magnet Axiom` [131], for example, integrates `Magnet AI`, a ML module capable of detecting grooming or luring conversations, identifying sexual content in chats, and flagging images showing drugs, weapons, nudity, or child abuse material. Similarly, `X-Ways Forensics` [215] incorporates Excire Photo AI [90], which can automatically classify photos, detect similar images, and perform facial recognition for known individuals, operating entirely offline to maintain case privacy. `Autopsy` [51], an open-source platform, provides a versatile environment for file recovery, timeline analysis, and keyword searches across a wide range of devices and file systems. Network evidence can be examined using tools like `Wireshark` [211] or `tcpdump` [201], which capture and decode network packets for further inspection. In cloud environments, platform-specific forensic utilities, APIs, and export mechanisms are often combined with conventional analysis tools to retrieve and interpret stored data.

The role of the DF consultant carries significant ethical, technical, and legal responsibilities that extend beyond mere technical competence. A DF consultant is not a lawyer, but an impartial expert whose duty is to provide an objective and scientifically grounded interpretation of digital evidence. Their task is to uncover and report facts as they are, without bias, omission, or manipulation, even when findings may not favor the client. Under no circumstance should the consultant alter, fabricate, or corrupt digital data; integrity and transparency remain paramount. A competent DF consultant must also accept only cases within their field of expertise, ensuring that their technical skills and experience align with the specific investigative scope, whether it concerns mobile

forensics, network analysis, or malware reverse engineering. Professional practice demands a formal contract, liability insurance, and, ideally, participation in a collaborative team where peer review strengthens the reliability of results. From a technical standpoint, the consultant must maintain precise documentation of every step, preserving a detailed chain of custody and recording all tools, commands, and parameters used. They must verify data integrity through hash calculations, keep verified backups, and ensure that every analytical conclusion can be replicated. Moreover, they should be capable of translating complex technical findings into clear, accessible language for legal practitioners and non-technical stakeholders. Precision, transparency, and methodological rigor are not optional but core ethical principles that define the professionalism and credibility of a digital forensic expert.

Today, DF is used across a wide range of contexts, extending far beyond its early association with criminal investigations. In the law enforcement domain, it serves as a cornerstone of cyber-crime investigations, enabling the reconstruction of events, the identification of suspects, and the presentation of admissible evidence in court. In corporate environments, it is employed for internal investigations, intellectual property theft cases, employee misconduct inquiries, and incident response following breaches. Within the cybersecurity sector, DF forms an integral part of threat hunting and post-incident analysis, allowing organizations to determine the scope of intrusions, understand the methods used by attackers, and implement preventive measures to avoid recurrence. It is also increasingly deployed in regulatory compliance, where organizations must demonstrate due diligence in protecting sensitive information and provide detailed audit trails in the event of data breaches. In civil litigation, forensic techniques can be used to support or refute claims involving contractual disputes, fraud allegations, or the misuse of digital assets. The rise of emerging domains such as autonomous vehicles, IoT-enabled environments, and blockchain systems has further expanded the application of DF, making it essential for accident reconstruction, device compromise investigations, and cryptocurrency transaction tracing.

The rationale for applying DF is behind its ability to uncover hidden, deleted, or obfuscated data and to establish a clear chain of events through scientifically validated methods. By preserving evidential integrity and ensuring reproducibility, it not only strengthens the legal standing of findings but also enhances operational resilience against future incidents. The systematic approach encompassing identification, acquisition, preservation, analysis, documentation, and presentation, ensures that investigations remain both legally defensible and technically robust, even when dealing with complex, multi-platform digital ecosystems.

However, despite its maturity as a discipline, DF faces several limitations and open challenges. One of the most pressing issues is the exponential growth of digital data, driven by the proliferation of connected devices, cloud platforms, and big data repositories. This not only increases the time and computational resources required for analysis but also raises difficulties in triaging evidence efficiently. Encryption technologies and privacy-preserving mechanisms, while essential for data protection, present significant obstacles for investigators, often making timely access to evidence impossible without legal intervention or specialized exploitation techniques. The widespread adoption of ephemeral messaging applications, volatile cloud storage, and decentralized networks further complicates evidence acquisition, as critical data may be transient or geographically dispersed across multiple jurisdictions. Emerging technologies present their own unique forensic challenges. Autonomous systems and IoT devices often use proprietary firmware and non-standard

storage formats, making acquisition and analysis highly device-specific and technically demanding. Blockchain transactions, while transparent, are pseudonymous and can involve complex chains of exchanges across multiple wallets and services, complicating attribution. Cloud forensics remains hindered by dependency on service providers for data access, inconsistent logging practices, and cross-border legal constraints. Additionally, the integration of Artificial Intelligence in both forensic tools and anti-forensic techniques introduces a dual-use dilemma: while AI can automate classification, pattern detection, and anomaly identification, it can also be leveraged by adversaries to generate synthetic data, automate evidence tampering, or deploy highly adaptive malware. Another fundamental challenge lies in the shortage of trained forensic professionals who possess both deep technical knowledge and a strong understanding of legal frameworks. Many forensic tools, especially those incorporating ML operate as proprietary black boxes, limiting transparency in methodologies and raising concerns over the reproducibility and admissibility of AI-assisted findings in court. The need for updated, universally accepted standards, especially for newer domains (e.g. IoT, industrial control systems, and cloud computing), remains urgent to ensure interoperability, reliability, and judicial acceptance across jurisdictions. DF techniques are also applied to malware analysis and detection because are extremely powerful in retrieving stealthy attacks such as malware using obfuscation, anti-analysis techniques or with adversarial behaviors.

In summary, DF is now indispensable in criminal justice, cybersecurity, corporate governance, and regulatory compliance, providing the means to extract, interpret, and present digital evidence in a legally defensible manner. Yet, the field must continually adapt to an evolving technological landscape, balancing investigative capabilities with privacy considerations, overcoming technical and jurisdictional barriers, and developing transparent, standardized methods to maintain trust in its findings.

3.3 Android Forensics

Android forensics is the branch of DF focused on the extraction, preservation, analysis, and presentation of evidence from devices running the Android OS. It involves examining both non-volatile storage, which contains the OS, APK, and user data, and volatile memory (RAM), which holds temporary system and application data. Investigators use a range of techniques, from physical acquisition (i.e. creating an exact bit-by-bit copy of storage media) to logical acquisition (i.e. retrieving data accessible through the file system). Specialized tools such as `Magnet AXIOM`, Cellebrite `UFED`, `LiME`, and `Volatility` are commonly employed to capture and analyze this data. Because Android devices vary widely in hardware and software configurations, and often include strong security measures like full-disk encryption and secure boot, forensic analysis frequently requires root access and the creation of custom memory profiles. The ultimate goal is to recover and interpret both persistent and transient digital artifacts while maintaining the integrity of the evidence for legal proceedings.

Different approaches have been developed depending on whether the target is non-volatile memory (such as solid-state storage) or volatile memory (RAM). Non-volatile storage contains the OS, applications, and user data, and persists even after the device is powered off. If the device is rooted, investigators can create a complete bit-by-bit image of the main storage device (often located at `/dev/block/mmcblk0`) using the Linux `dd` command. This physical acquisition preserves every sector of the storage medium, including allocated, unallocated, and hidden areas, ensuring the

captured image is identical to the original and suitable for evidential purposes. If root access is unavailable, investigators often turn to specialized forensic tools such as Magnet `AXIOM`, Cellebrite `UFED`, `Autopsy`, or `X-Ways` Forensics, which can perform either physical or logical acquisition. Logical acquisition is faster and limited to data visible in the file system, making it less comprehensive and potentially missing deleted or hidden artifacts.

RAM acquisition focuses on capturing the device's temporary working state, which disappears once the device is powered off. Investigators may opt to dump the entire memory using tools such as `LiME` or target a specific process with tools like `fridump`. Analyzing RAM requires forensic frameworks such as `Volatility` or `Rekall`, which can parse running processes, loaded modules, network activity, and other transient artifacts. For these tools to interpret Android RAM correctly, a memory profile describing the device's OS, kernel, and memory structure is needed. Because Android is based on the Linux kernel and varies significantly across devices, prebuilt profiles are rarely available. They must be created manually or downloaded from trusted sources, a process that typically requires root access to extract the kernel image from the boot partition.

In modern devices, DF acquisition faces additional challenges from security features like full-disk encryption, hardware-backed key storage, and verified boot. These measures can prevent direct access to raw data even with physical imaging techniques, sometimes requiring investigators to extract encryption keys from RAM before they are lost or to employ specialized hardware methods such as JTAG or chip-off. Maintaining the chain of custody and ensuring data integrity remain central to the process, with physical acquisition offering the strongest guarantees for evidential purposes. In practice, combining both storage and RAM acquisition methods provides the most complete picture, allowing investigators to recover persistent data from the storage medium and transient state information from memory for a thorough DF analysis.

Chapter 4

Current Research and Approaches on Android Malware Detection

This chapter describes the current literature for the Android malware analysis and Forensics, with a focus on Memory Forensics. Everything is then oriented on the application of AI, in particular describing current AI-based techniques and how AI can help in the presented problem.

4.1 Android Malware Detection

The current state of the art in Android malware detection combines classical *static analysis* to extract semantic and syntactic properties, (e.g. manifest permissions, API-call graphs, control-flow structures, opcodes and embedded strings) without executing the app, with *dynamic analysis* to monitor behavior through API traces, network traffic, and system calls during execution. *Static analysis* inspects an application's bytecode, manifest, and resource components without executing the program. This method encompasses several granular techniques: *(i)* syntactic analysis [20, 35, 151], which extracts textual and structural features such as permissions, intents, strings, opcodes, or imported APIs; *(ii)* semantic analysis [37, 96, 209], which reconstructs control-flow and data-flow graphs to infer higher-level behavior and dependency relations between components; and *(iii)* metadata analysis [67, 231], which considers certificates, signing information, and package provenance. Systems such as `DREBIN` [35] established the viability of large-scale static learning by vectorizing features derived from manifests and API calls into interpretable linear models. Subsequent works expanded toward graph-based representations, permission clustering, and DL over byte sequences or opcode embeddings [37]. *Static analysis* is lightweight and scalable, making it indispensable for app-store screening and large-scale triage. However, it is inherently limited against obfuscation, code packing, reflection, and dynamic code loading, which can conceal malicious payloads or modify execution logic at runtime.

Dynamic analysis executes the application in an instrumented environment (typically an emulator, VM, or physical device) to observe runtime behavior. Depending on the level of instrumentation, dynamic systems can capture: *(i)* system-call traces [73] that record kernel-level interactions; *(ii)* API-call sequences [135, 186, 199] reflecting application behavior at the framework layer; *(iii)* network and file I/O [234] for identifying exfiltration or command-and-control activity; and *(iv)* user-interface events and sensors [183] for behavioral triggers. Tools such as `DroidScope` [220]

and CopperDroid [199] reconstruct high-level semantics from virtual machine introspection (VMI), correlating OS-level and Dalvik-level events into comprehensive behavioral profiles. Behavioral abstraction frameworks like MaMaDroid [135] further model API sequences as Markov chains to capture patterns of malicious logic independent of specific API versions. Despite providing concrete runtime evidence, dynamic analysis faces key challenges: coverage limitations due to incomplete execution paths, dependence on trigger conditions or user interactions, and the widespread use of anti-analysis [137, 207] measures such as emulator and instrumentation detection, timing checks, and logic bombs that activate only in specific contexts [172]. Hybrid analysis seeks to integrate static and dynamic perspectives into a unified detection pipeline. Previous hybrid detection mechanisms combined static extraction (permissions, API graphs) with dynamic runtime features (system calls, network activity), feeding them into ensemble or multi-view learning models [133]. Limitations include coverage gaps, trigger dependence, and widespread anti-analysis (emulator/instrumentation/timing checks). Modern approaches extend this principle by embedding runtime observables directly into DL representations [95]. For instance, graph-based neural networks can encode both static control-flow edges and dynamic API traces, learning correlated behavior across layers [216]. Another prominent subfield is *memory-aware analysis*, in which volatile memory (RAM) is treated as a high-fidelity source of behavioral evidence [113, 134]. During execution, malware frequently decrypts, unpacks, or dynamically loads payloads into memory that never appear in the static APK. By acquiring per-process or system-wide memory snapshots, analysts can recover hidden code, live strings, and native payloads that evade both static and dynamic tracing [113]. Memory-centric systems demonstrate that examining process memory at runtime yields stronger ground truth and enables detection of in-memory and ephemeral threats.

Building on these foundations, recent research explores multimodal and deep representation techniques that generalize beyond manual feature engineering. Binary-to-image and memory-to-image encoding maps raw byte sequences or memory segments into grayscale or RGB matrices, enabling CNN or vision transformers to capture spatial correlations between opcodes and data regions [29, 68, 84]. Frameworks such as DexRay[68], Didroid[160], and cRGBMem[29] achieve high accuracy and scalability, though their performance depends on the choice of encoding (1D vs. 2D, grayscale vs. RGB). Beyond byte-level images, transformer-based sequence models and graph neural networks (GNNs) are applied to represent control-flow graphs, API dependencies, and permission hierarchies as embeddings that preserve contextual semantics[]. These deep architectures learn discriminative latent features capable of generalizing across families and obfuscation variants. With DL models increasingly employed in malware detection, *explainability* has become critical. Techniques such as Grad-CAM, LIME, and SHAP [187] enable analysts to visualize which bytes, API calls, or instruction regions influence model decisions, providing transparency for forensic validation. At the same time, adversarial ML exposes detectors to new risks: crafted perturbations or repackaged can induce misclassification, underscoring the need for adversarially robust training [70]. Robust models now incorporate input sanitization, feature randomization, and ensembles of orthogonal modalities (static with dynamic with memory), reducing susceptibility to single-vector attacks.

Despite their maturity, traditional techniques suffer when faced with modern evasion tactics such as obfuscation, repackaging, dynamic code loading, and adversarial perturbations that mislead learning-based detectors [35, 63, 70].

Repackaging (i.e. injecting malicious logic or native libraries into popular apps and resigning

them) [137, 166] undermines detection and enables long-lived stealth unless provenance and structural consistency are verified or runtime loading is monitored. Obfuscation and program transformations (e.g. identifier renaming, string encryption/fragmentation, control-flow flattening, opaque predicates, native packers, reflection and dynamic class loading) are widely observed in the wild and defeat brittle feature extractors; robust detectors therefore rely on semantic or behaviorally grounded representations and on deobfuscation/unpacking stages [122].

Anti-analysis techniques form a broader defensive layer against automated investigation, ranging from anti-static, anti-dynamic and anti-forensic to symbolic-execution evasion. *Anti-static* methods include code encryption, multi-stage loaders, resource-based or native encryption keys, and self-modifying DEX segments that decrypt only in memory [109, 149, 185]. *Anti-dynamic* strategies detect instrumentation frameworks (e.g.`Frida`), debug hooks or timing anomalies, and abort or alter behavior under observation [207]. Malware further exploits anti-sandbox heuristics (i.e. checking sensors, device identifiers, filesystem artifacts, or network endpoints) to distinguish virtual from physical devices [27, 137, 168]. Some samples perform *anti-symbolic evasion* [38, 204], inserting opaque predicates and large numerical loops that explode symbolic-execution search space, or using randomization to generate unique control paths. Finally, *anti-forensic* tactics [26, 34, 150] target post-mortem investigation by deleting traces, encrypting logs, wiping shared preferences, or destroying payloads on detection of forensic tools.

These evasion categories share a goal: reducing observability by the analyst. Consequently, modern pipelines integrate counter-evasion strategies such as *(i)* on-device execution and large-scale, sensor-faithful emulation to minimize environmental gaps [73]; *(ii)* extended execution with randomized user-interaction replay [80]; *(iii)* multi-context instrumentation and volatile memory dumps to capture decrypted or unpacked code [113, 134]; and *(iv)* dynamic taint and API-trace correlation to recover hidden semantics [37, 73, 209]. Advanced sandboxes implement deceptive environments, simulating telephony activity, GPS drift, accelerometer noise, or benign background processes to defeat sandbox detection [33, 168]. Anti-forensic resilience is addressed through immutable logging and snapshotting frameworks that record volatile artifacts before self-destruction [26, 34]. Together, these measures reduce the asymmetry between adversarial evasion and analyst observability [134, 199, 220].

Adversarial ML exposes learned detectors to both feature-space [63, 70] and problem-space [97] attacks; the latter apply feasible APK transformations that preserve functionality while altering extracted features, necessitating adversarial training with realistic transforms, ensembles of orthogonal signals, input normalization, and threat models that respect Android build/execution constraints [70]. Temporal drift [64, 140, 141, 171] (i.e. evolving APIs, libraries, developer practices, and malware families) causes evaluations to overestimate deployed performance; current best practice emphasizes time-split protocols (train older and test newer) and continuous monitoring of performance decay. For Cyber Threat Intelligence (CTI), family detection/clustering fuses static code similarity, dynamic behavior fingerprints, and learned embeddings (graphs or byte-images) to separate polymorphic variants from shared third-party libraries. Finally, stealthy attacks combine the above tactics: low-and-slow logic, memory-only payloads, native-library abuse, and steganographic media; here, monitoring the loading stage and pairing dynamic traces with volatile-memory snapshots is decisive.

To overcome these limitations, contemporary research focuses on hybrid pipelines that integrate

static and dynamic views and on the use of ML and DL to automatically learn discriminative patterns from large-scale data without handcrafted features [108, 116]. Modern Android malware detection is converging toward holistic, cross-layer frameworks that fuse static, behavioral, memory, and multimodal representations; harden against repackaging, obfuscation, emulator detection, and adversarial manipulation; employ explainability for analyst trust; and integrate vulnerability/exploitability reasoning so detectors prioritize threats with realistic impact [35, 68, 134, 135]. Static analysis remains essential for scalability and explainability; dynamic and memory-aware analyses provide behavioral and ground-truth evidence; and hybrid or multimodal pipelines unify these signals within deep-learning frameworks capable of detecting stealthy, dynamically loaded, and adversarially perturbed threats. This methodological convergence marks a shift toward cross-layer detection architectures that integrate static structure, runtime context, memory semantics, and interpretability, advancing resilience against obfuscation, repackaging, and the continuously evolving Android threat ecosystem.

Beyond bytecode, the Android attack surface increasingly involves native libraries (JNI and NDK) [75, 87] and multimedia payloads [56–58]. Vulnerabilities in native ELF components may act as hidden entry points for code injection and privilege escalation, motivating analyses that correlate Dalvik and native semantics to estimate exploitability and reachability. Likewise, Android *stegomalware*[56–58] (i.e. applications embedding concealed payloads in images, audio, or video resources) demands multimodal inspection pipelines. Recent frameworks perform *(i)* runtime extraction and decoding of media, *(ii)* feature-based analysis of the decoding logic, and *(iii)* correlation of memory snapshots with media artifacts to identify transient, in-memory payloads invisible to traditional scanners [69, 188, 189]. Studies such as the 2025 *Analysis and Detection of Android Stegomalware: The Impact of the Loading Stage* [188] demonstrate that decisive indicators of malicious intent often appear only during the decoding and loading phase rather than in the static APK. Recent works extend detection into the *runtime memory* domain, where process or system RAM is analyzed to reveal unpacked code, decrypted payloads, and in-memory resources invisible to static or sandbox analysis [29, 114]. This memory-aware direction enables discovery of payloads that materialize only after the application starts, providing stronger ground truth for detection and forensic validation.

Despite rapid progress, significant gaps remain. *(i) Data and evaluation.* Public Android datasets are biased and noisy (e.g. VT-derived labels), with limited longitudinal coverage; random splits overestimate performance under *temporal drift.* Time-split evaluations, library decontamination, and family-aware protocols are still inconsistently applied. *(ii) Robustness and generalization.* Models remain vulnerable to problem-space transformations (repacking, string/CFG obfuscation, native loaders), out-of-distribution inputs, and adversarial manipulation; robustness beyond feature-space attacks is underexplored. *(iii) Reachability and exploitability.* Many detections arise from third-party libraries or dormant code whose runtime reachability is unverified, inflating false positives; integrating path/trigger analysis and native/Dalvik cross-flow is an open need. *(iv) Runtime realism and anti-analysis.* Coverage is constrained by triggers and user interaction; emulator/VM artifacts, sensor stubs, and timing side-channels enable sandbox evasion, while memory-only payloads challenge trace-based systems. *(v) Native/JNI and multimodal payloads.* Visibility into ELF/JNI components, JIT/AOT artifacts, and steganographic media loading remains partial; correlating media decoding with time-aligned memory snapshots is promising but not yet standardized. *(vi) Deployment constraints.* On-device inference faces energy/latency/privacy lim-

its; telemetry restrictions and legal constraints inhibit data collection for continuous learning. *(vii) Reproducibility and forensic soundness.* Memory-aware pipelines lack widely accepted acquisition protocols and benchmarks; ensuring chain-of-custody and providing auditor-grade explanations (beyond Grad-CAM/LIME/SHAP) is essential for forensic defensibility. *(viii) Model lifecycle.* Concept drift, model aging, and data poisoning risks necessitate monitored updates, rollback strategies, and defense-in-depth (ensemble, multi-view, and invariant features).

4.2 Android Forensics for Threat Analysis

Android forensics has matured into a core sub-discipline of DF, concerned with the systematic extraction, preservation, and interpretation of evidentiary data from Android devices, strictly related by the rapid OS evolution, vendor fragmentation, and multi-layered security hardening. Methodologically, data acquisition is traditionally divided into logical, file-system, physical, volatile-memory, network, and cloud-based techniques, each offering different trade-offs between data completeness, invasiveness, and forensic soundness.

Logical acquisition exploits ADB interfaces, synchronization APIs, or system backup mechanisms to collect user-accessible data such as call logs, SMS/MMS, contacts, and application databases [94]. This approach, implemented in commercial suites including Cellebrite `UFED`, `Oxygen` Forensic Detective[153], enables analysis of non-rooted devices through automated permission negotiation and structured database parsing [39, 206]. Open-source frameworks like `Andriller` or `Autopsy` support similar workflows [31]. Logical extraction is the least invasive, as it operates within the system's own security model, ensuring forensic soundness and minimal alteration of the device. However, its scope is inherently limited, it cannot recover deleted files, volatile memory artefacts, or system-level traces, and it often omits encrypted and sandboxed data [206].

File-system acquisition extends analytical depth by reconstructing complete directory hierarchies, permissions, and timestamp metadata from internal and external partitions. Using frameworks such as `Autopsy`, Magnet `AXIOM`, and ALEAPP[48], investigators can access app caches, configuration files, shared preferences, and event logs stored in `/data`, `/system`, or `/sdcard`. These artifacts are instrumental in timeline reconstruction and cross-application correlation, allowing reconstruction of user behavior and app interactivity across time.

The *physical acquisition* layer generates bit-for-bit images of NAND or eMMC memory that include deleted, hidden, and encrypted data. Tools such as the native `dd` utility, the `LiME` kernel module, or hardware-assisted JTAG, chip-off, and ISP (In-System Programming) procedures provide direct access to low-level storage [71, 120, 223]. However, the progressive implementation of Secure/Verified Boot, Full-Disk Encryption (FDE), File-Based Encryption (FBE), and Trusted Execution Environments (TEE) significantly restricts these methods [40]. In most modern devices, physical imaging requires bootloader unlocking or temporary rooting. Such actions modify the device state and compromise evidential integrity. Consequently, forensic research has shifted toward selective physical and rootless live acquisitions that prioritize forensic soundness over full access, such as recovery partitions or kernel-level modules signed with OEM keys.

Volatile-memory acquisition has emerged as a crucial complement to storage-based forensics, addressing the evidentiary gap left by encryption and runtime obfuscation. Memory forensics captures dynamic, volatile artifacts (e.g. running processes, cryptographic keys, network buffers, injected

payloads, and decrypted app data) that disappear once the device is powered off. Tools like `LiME`, and `Fridump` capture full or process-scoped RAM snapshots, later analyzed using `Volatility` or `Rekall` [139, 163, 196, 208, 222]. The Just-in-Time Memory Forensics (JIT-MF) framework introduced by Bellizzi *et al.* [40–43] allows timely evidence capture from stock, non-rooted Android devices, mitigating the need for persistent root access. Similar `ptrace`-based and `Frida`-instrumented approaches facilitate per-process memory acquisition, particularly effective in forensic examination of encrypted messengers and financial apps [222]. Dynamic instrumentation frameworks (e.g.`Frida`) extend these capabilities by injecting runtime hooks to monitor API calls, inter-process communication (IPC), and decrypted memory objects [222]. When synchronized with memory snapshots, such methods expose volatile runtime states, reflective loading, and polymorphic malware routines invisible to static inspection. Memory forensics frameworks such as Volatility have been adapted to Android environments to parse kernel structures, identify processes, and recover artifacts from Dalvik or ART runtimes [208]. Recent research has expanded this domain through advanced encoding-based analysis techniques, transforming binary memory regions into images, audio representations, or symbolic embeddings to facilitate detection of obfuscated or steganographic content through machine learning models, as seen in works like DexRay and CRGBMem [29, 68].

Complementing these device-centric methods, network and traffic acquisition has become increasingly relevant in Android investigations, as many mobile applications offload data to remote servers or employ encrypted communication channels. *Network forensics* in Android involves capturing and analyzing live or stored traffic using tools such as `tcpdump` and `PCAPdroid`, while analysis platforms like `Wireshark`, `Zeek`, and `Scapy` facilitate packet-level inspection and flow reconstruction [101, 181, 201, 211, 227]. This dimension is particularly important in modern investigations involving cloud-connected applications or malware using covert network channels [65, 71].

Despite significant advancements, the field continues to face major challenges. Device fragmentation and rapid OS updates complicate kernel module compatibility and tool stability; secure storage and encryption mechanisms restrict access to critical artifacts; and the increasing use of anti-forensic techniques (e.g. ranging from obfuscation and sandbox evasion to ephemeral data handling) requires adaptive strategies and constant tool evolution [210]. Moreover, the growing reliance on AI and automation in forensic analysis introduces both opportunities and concerns regarding transparency, explainability, and evidential admissibility [44, 210].

Overall, the current state of Android forensics reflects a transition from tool-centric practices toward more systematic, intelligent, and context-aware methodologies. The discipline increasingly emphasizes reproducibility through digital twin environments, multi-modal evidence correlation, and AI-driven analytics that can identify patterns across heterogeneous data sources. This convergence of acquisition, automation, and analytical depth positions Android forensics as a critical frontier for cybersecurity, criminal investigations, and digital trust in an ecosystem that is both ubiquitous and continually evolving.

4.2.1 Android Memory Forensics for Malware Analysis

Android memory forensics integrates a broad and evolving set of acquisition, extraction, and analysis techniques aimed at recovering volatile runtime evidence, essential for both malware detection and the reconstruction of transient artifacts such as in-app chats, decrypted payloads, credentials, and ephemeral communication data. Acquisition strategies vary in scope and invasiveness: phys-

ical methods such as chip-off and JTAG/ISP provide bit-level completeness but are destructive and rarely feasible on encrypted devices [52, 197]; controlled-boot and forensic-boot environments enable trusted memory capture when bootloaders are unlocked [121], while live kernel captures like `LiME` remain the dominant open-source solution for active devices, though they require kernel-specific compilation[19]. Alternatives such as `Memfetch` [226], `AMExtractor` [221], and `Frost` [143] attempt to overcome kernel-module or cold-boot limitations, yet face compatibility issues on modern Android. Targeted process extraction frameworks (e.g.`Frida`/`Fridump`) [85, 86], heap-dump approaches (Java/ART hprof snapshots) [228], and runtime instrumentation (JIT-MF) [40, 42] provide fine-grained visibility into specific apps or runtime states but are limited in coverage and timing precision.

Once memory is captured, extraction transforms raw bytes into semantic entities. Frameworks such as `Volatility` and `Rekall` parse kernel and process structures (e.g. task lists, virtual memory areas, file descriptors, and socket tables) but require OS specific profiles that are hard to generate for custom Android kernels; `Autoprofile` automates this process through kernel symbol inference [154]. Runtime reconstruction tools like `DroidScraper` [28] and Java-heap analysis frameworks [228] rebuild Dalvik/ART object graphs, revealing classes, strings, and message buffers, while others focus on native heap parsing to identify injected libraries or dynamically loaded code segments [82, 232]. Binary carving, entropy profiling, and string/regex searches remain standard for recovering file fragments, keys, and configurations [190, 203], whereas statistical and ML-based methods classify regions of interest by entropy, n-gram distribution, and byte frequency to distinguish packed or encrypted sections [113].

For malware analysis, reconstructed code and artifacts feed hybrid workflows that merge static and dynamic reasoning: disassembly of carved native code, decompilation of Dalvik/ART bytecode, reconstruction of API-call graphs, and correlation of in-memory API traces or open sockets with known C2 patterns [63]. ML and DL approaches, such as binary-to-image encoding [145] and representation learning on opcode or byte sequences [133], have been adapted to volatile data to automatically prioritize suspicious memory slices and detect packed payloads or reflective loading behavior. For chat and messaging recovery, memory forensics leverages managed-heap reconstructions and SQLite page carving to extract plaintext message objects, cached attachments, and conversation metadata, even in end-to-end encrypted apps where decrypted content resides only briefly in memory [41, 52, 102].

Despite progress, major practical challenges remain: device and kernel fragmentation, secure boot and `SEAndroid` enforcement, transient garbage collection and JIT churn that overwrite artifacts, hardware-backed key storage (TEE/Keystore), and anti-forensic countermeasures like memory scrubbing or runtime integrity checks [22, 179, 193]. Many tools are unmaintained and cannot support recent Android releases, and ML pipelines still lack diverse, annotated volatile-memory corpora for adversarially robust training. Consequently, modern research emphasizes hybrid, multi-vector acquisition, automated profile generation, event-triggered snapshotting, and explainable AI-assisted classification coupled with rigorous chain-of-custody documentation to preserve forensic soundness while exposing in-memory decrypted payloads, runtime-only malicious behaviors, and transient user communications that remain invisible to static or network-based analysis.

Despite these advances, major limitations persist. From a technical perspective, device and kernel dependency remains a critical barrier: memory-acquisition tools require recompilation for specific

kernel builds, and secure boot mechanisms (e.g. Verified Boot, TEE/Keystore) block direct access to physical memory, limiting forensic completeness. Data dependency is equally restrictive as many tools capture only subsets of RAM (e.g. Dalvik/ART heap or target process space), excluding native buffers, or kernel allocations, resulting in incomplete evidence. Volatile data further complicates analysis, as garbage collection, JIT recompilation, and thread scheduling overwrite evidence within milliseconds, while encryption-in-use and anti-forensic behaviors (e.g. memory scrubbing, runtime packing, reflective code loading) deliberately conceal malicious content [22, 179, 193]. Methodologically, profile generation and validation are cumbersome, requiring deep OS knowledge and manual adaptation to evolving Android versions. Machine learning pipelines, though promising, face challenges related to dataset scarcity, adversarial robustness, and cross-device generalization; current models are often trained on synthetic or desktop memory corpora that poorly reflect the Android runtime environment. Furthermore, tool obsolescence is endemic as most open-source or academic frameworks (e.g.`DeepMem`, `Memgrab`, `AMExtractor`) are not maintained beyond a few Android generations, reducing reproducibility and community validation.

4.3 Artificial Intelligence for Android Malware Forensics

The integration of AI into Android malware forensics represents a transformative advancement in DF, addressing the growing complexity, scale, and evasiveness of modern Android threats. Traditional forensic workflows (i.e. relying on static signatures, manual reverse engineering, and rule-based anomaly detection) are increasingly inadequate against polymorphic, obfuscated, and context-aware malware that continuously evolve to bypass conventional defenses. AI-driven approaches, encompassing ML, DL, and graph-based reasoning, enable the automatic discovery of subtle behavioral and structural patterns that are often imperceptible to human analysts. By learning from large datasets of benign and malicious applications, AI models can generalize beyond known attack vectors to detect zero-day samples and infer malicious intent even under obfuscation. In forensic practice, AI enhances every stage of the investigation pipeline: it supports automated triage and clustering of samples, dynamic behavior profiling, semantic reconstruction of code flows, and correlation of artefacts across devices and memory snapshots. Moreover, AI techniques can prioritize evidentiary relevance, reducing analyst workload and accelerating incident response in large-scale digital investigations. As Android's ecosystem continues to fragment across versions, vendors, and architectures, the adaptability and scalability of AI-based forensic tools make them indispensable for maintaining accuracy, reproducibility, and timeliness in the face of rapidly evolving mobile threats.

Chapter 5

Evaluating the Exploitability of Android Native Code

Android applications are widely analysed in terms of malware detection and vulnerability assessment. One less common analysis involves the study on Native Code vulnerabilities, i.e. C/C++ libraries used in Android to interact with native components, frequently imported from third-party developers and inheriting C/C++ common vulnerabilities such as format string, buffer overflow. If such vulnerabilities are exploited, an attacker can have access to sensitive private data or inject malicious code as C/C++ vulnerabilities permit direct access to the main memory (RAM). In this chapter, I leverage the limits of current native code vulnerabilities analysis and answer to the first research question of this thesis ***How to access the content of the RAM?***. I first construct the knowledge base and dataset for native code vulnerabilities by developing a methodology to quickly evaluate the risk of a known vulnerability Common Vulnerability Exposure (CVE) in an app. Secondly, I define a methodology on how to call such vulnerable function from the Java code and subsequently which input to send. Because of the lack of existing public datasets, I deeply define the methodologies to use but restricted tests have been made. In particular, I define how and why AI-based algorithms can help in the Android native code vulnerability assessment. The main contributions of this Chapter are threefold: *(i)* the definition of a risk-based methodology to assess vulnerabilities; *(ii)* a risk score assigned to the detected vulnerabilities in real world applications; *(iii)* a graph-based approach to call the vulnerable function starting from the Java/Kotlin code; *(iv)* a fuzzing methodology on the reachable vulnerable function to find the input triggering the detected vulnerability. In particular, this Chapter asks *How to exploit Native Code vulnerabilities and reside in RAM?*

5.1 Assessing Native Code Vulnerabilities

The growing complexity of Android applications has made Native Code a critical yet vulnerable component of the mobile software ecosystem. Although most Android apps are primarily developed in Java or Kotlin, performance-critical functionalities (e.g. multimedia decoding, image processing, cryptography, and low-level hardware interactions), frequently rely on native libraries implemented in C/C++. These languages offer speed and flexibility but inherit long-standing memory-safety issues if not properly managed during development, including buffer overflows, use-after-free bugs,

and improper pointer handling. Exploiting such flaws can allow attackers to read or modify sensitive data stored temporarily in clear text in the RAM, cause denial of service, or even execute arbitrary code, leading to full device compromise. Despite their impact, vulnerabilities in Native Code remain less systematically analyzed than those in the managed Java/Kotlin layer, creating a persistent blind spot in Android security.

Existing approaches to analyze and mitigate Native Code vulnerabilities are often resource-intensive and difficult to scale. Deep static analysis, dynamic tracing, symbolic execution, and binary similarity matching can achieve high accuracy but require expert knowledge, significant computational resources, and careful configuration. Moreover, real-world Android applications frequently embed third-party native libraries making dependency tracking and version identification challenging. Even well-documented vulnerabilities may persist unpatched for years, as developers may not be aware of their inclusion or lack sufficient visibility into the libraries' provenance. While a few tools, such as `Librarian`[30], have attempted to automate Native Code analysis, these typically provide binary outputs ("vulnerable/not vulnerable") without a nuanced estimation of risk or prioritization of vulnerabilities. Consequently, the security community still lacks lightweight, automated, and scalable approaches for assessing the security posture of Native Code within Android applications. A research question arises automatically: ***Is it possible to define a score to prioritize vulnerabilities assessment?***

The identification of vulnerabilities in Android native code is a relatively new research area with few existing studies. `Librarian` [10] analyzed vulnerabilities in the top 200 Android apps using a binary similarity algorithm (`bin2sim`) to identify libraries and detect vulnerabilities through a whitelist approach. While accurate, it is computationally intensive and does not provide risk scoring. Other relevant studies address reachability analysis, such as `DroidReach` [45], which uses symbolic execution to determine whether vulnerable functions are actually called, though its complexity limited direct adoption in this work. Ruggia *et al.* [170] introduced a static ML-based method to detect suspicious native code patterns in malicious apps, showing how native code complicates malware analysis. Early efforts like Fedler *et al.* [77] aimed to mitigate root exploits, though these methods became outdated as native code use increased. Popular tools like `MobSF`, `Qark`, and `SEBASTiAn` can scan APKs but offer limited coverage of native components. Brant *et al.* [47] highlighted shortcomings in Android's security bulletin testing coverage, emphasizing the need for better patch validation. `LibRadar` [130] and Li *et al.* [123] improved library identification through API-based and metadata-driven analyses, though these are unsuitable for C-based native code. Other approaches such as `GoingNative` [22], `NativeGuard` [193], and `AppCage` [233], focused on sandboxing native code, while `Ndroid` [218], `DroidNative` [25], and `AdDetect` [144] explored the interaction between Java and Native Code for malware detection. In summary, existing works generally achieve high accuracy but at significant computational cost, often relying on large datasets or ML training.

To address this gap, this work introduces a novel methodology for risk estimation of vulnerabilities in Android Native Code, emphasizing speed, scalability, and integration within early development workflows. The proposed approach does not aim to replace high-precision, resource-heavy analysis but rather to complement it by providing an efficient early-warning system. It enables quick identification of libraries likely to contain known vulnerabilities, assigns a qualitative risk score to each app, and offers actionable insights for developers and security analysts before re-

lease. This methodology aligns with the growing adoption of Software Bill of Materials (SBOM) practices, which promote transparency and accountability in software supply chains by tracking components and their associated security risks. A key reason for this neglect is the difficulty and cost of traditional vulnerability analysis for Native Code.

This methodology unfolds in several stages as shown in Figure 5.1. Given an APK, the native library is extracted from its `lib` directory. As the compiled library is an ELF file, specific reverse engineering tools are needed and techniques to extract data for vulnerability detection. Such data is the list of functions and the product name with the version to which the library belongs. Once gathered such information, a match of this result with a database of known vulnerabilities is made. The database is purpose-specific, containing for each CVE [6] a field with the affected vulnerable versions and functions. At the moment of this writing, there is no publicly available database for this purpose, and with this specific structure, a system can easily access and read it. To construct the database, a list of N products is needed (otherwise it is very hard to consider every released library). The product matching process is a *whitelist approach* that assumes that none of the Android app's developers has changed the library name (keeping the real one employed during compilation time in the NDK). Once there is the match between the library under analysis and the database, a risk assessment score can be assigned to the found CVE in the analyzed library. Our purpose is to give a risk value to each library (according to the risk of each CVE found) and, consequently, to each app to provide an alarm to developers.

The first stage is the construction of the custom CVE database to check if the extracted data from the analyzed library have been declared in a published vulnerability list. This database is constructed locally to ensure high-speed lookups, reducing the latency of querying public online databases and collecting all useful information together. Hence, the local database is a dump of data contained in the online selected website of CVEs (e.g.`NVD` [11]) and with fields organized according to the aim of this research. While public repositories like `NVD` or `Mitre CVE` are rich in data, their human-readable vulnerability descriptions vary greatly in structure. To make the data machine-usable, NLP techniques have been applied (specifically using the `nltk` Python library) to parse CVE descriptions and extract three key attributes: the affected product name (e.g.`OpenSSL`, `SQLite`, `FFmpeg`, etc), the vulnerable versions, and the names of any explicitly mentioned vulnerable functions. The *product name* is easily matched with one of the N selected products, and if one is mentioned inside the description, the CVE has been found in that specific product. This can be further confirmed by searching all vulnerabilities by product name in the public database. The *version of the library* affected by the CVE is always a number that may come after the word *version*, the product name (i.e. affected product version is only the one found in the description), or preceding words with the meaning of *after* (e.g. after, following, successive; i.e. every product version higher than the one found in the description is vulnerable) or *before* (e.g. before, prior, earlier; i.e. every product version lower than the one found in the description is vulnerable). To recognize functions' names I relied on how programmers typically recognize function names and give names to functions. Then, by looking at some CVE descriptions, I noticed that the function is never declared as, for example, *function F*, represented uniquely by its name. The function name usually contains specific symbols (i.e. _, ::, (), etc.). Moreover, if the name is made up of more than one word, it is camel-cased. As an example, `makeSum` is made up of two words: *make* and *Sum*. In other cases, the function is not found, which means that the whole product version is vulnerable,

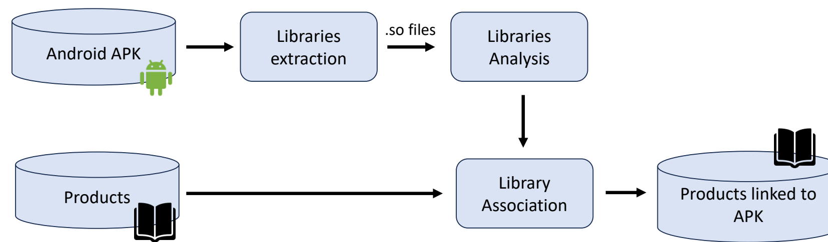

Figure 5.1: Workflow of the approach to extract, analyze, and associate native libraries

but no description of the vulnerable function is provided.

After having constructed the database, each APK is analyzed. First, the APK file is unpacked and locate the lib directory, which contains native library binaries compiled, hence the *.so* ELF file. From each library identifying information has been extracted, i.e. the product name, version number, and a list of function names—using reverse-engineering techniques and automation tools like the `pwntools` Python framework [12]. Product (i.e. the productory vendor library name such as `OpenCV`, `OpenSSL`, etc) and version information typically reside in `.rodata` sections as strings, while function symbols are found in the ELF symbol table. The extracted information is then compared against a custom CVE database. Library extraction is the first step of the analysis part. It is done by unzipping `.so` files from the `lib` directory of each APK. Indeed, libraries are compiled according to different ABI and saved inside the application.It is possible to extract libraries only for specific architectures or, alternatively, to analyze libraries for all available ABI. In this work, all ELF files have been considered from `armeabi-v7a` directory, and if not available, looked at `arm64-v8a` or `x86_64` because they are the most popular architectures. Library analysis is the part where I need to extract from the library the data for vulnerability detection. In particular, I need to know the list of functions in the library and the product name with its version to which the library belongs. This step can be done with different reverse-engineering tools, such as `Ghidra` and its Python extension for automation. Typically, the product name and its version can be retrieved in the strings section (`.rodata` section), while the defined functions in the ELF file can be retrieved from the Symbol Table section. In this work, `pwntools` has been used, a popular Python framework for binary analysis and exploitation. Specifically, its `ELF` module has been employed, which allows the analysis of ELF files from which the strings and the list of functions are extracted. The version is taken from the result of `strings` Linux utility[1]. At the same time, the functions are found inside the Symbol Table of the ELF without considering `.got` and `.plt` sections. A difficulty comes when binaries are stripped[2] where not all function names are available, or the names are unrecognizable. is the part where each ELF file is associated to at least one of the selected N products. This is a crucial task: if each ELF file is not linked to its related product, it is impossible to determine if the ELF file contains vulnerabilities, hence assigning a risk assessment score. Different works used binary similarities techniques or ML algorithms; however, I decided to apply a simple identification

[1]`strings` tool in Linux is capable of retrieving all printable sequences of characters from the .rodata section of an ELF.

[2]A stripped binary is a binary without some debugging symbols and so with a lack of data.

algorithm because it is widely known that every product uses a clear and unique syntax in strings and function names. For example, in `OpenCV`, strings like *General Configuration for OpenCV v.n* to declare the version and *xxxx_ cv_ xxxx* in the function names can be clearly found. For this reason, if these syntaxes in strings and functions are found, a link with the analyzed ELF file to a product is made. Some binaries can be linked to multiple products due to imported libraries. Indeed, in a compiled library, there can be traces of the *primary* library and the *secondary* libraries (i.e. the ones employed by the primary). In this case, all retrieved products' ELF files were considered, since a clear distinction between primary and secondary libraries is often difficult to achieve in practice, an issue also highlighted by Borzachiello *et al.* [45].

After matching libraries to known vulnerable products, the risk assessment algorithm is applied. I developed a risk assessment algorithm to give a risk value to each library (and consequently to each app) to provide an alarm to developers. Even though a CVE is present in the analyzed libraries within an ELF, it is not possible to be 100% sure that it is exploitable due to stripped binaries, imported products, and the simplified library identification approach employed in this work without considering the reachability problem. For this reason, the problem is approached in *probabilistic terms*, and I developed a semi-quantitative risk assessment algorithm. The risk in Equation 5.1 follows the ISO 27005: [17] framework.

$$risk = threat * impact * vulnerability \tag{5.1}$$

A *threat* corresponds to an action that negatively impacts the device. Hence, the threat factor can be associated with the ease with which an attack can be carried out. To quantify it, the Common Vulnerability Scoring System (CVSS) [7] exploitability value is used since it has a similar definition, regardless of the attacker's capabilities. The *impact* is the damage caused to the system if the vulnerability is exploited. It can be quantified with the CVSS impact value without considering the architecture of the victim device. The *vulnerability* can be identified as the CVE itself, with an ideal value of 1 (which refers to its presence). When a CVE is identified and confirmed to be present and exploitable, the term vulnerability is assigned a value of 1. Instead, when the CVE is not present or not exploitable, the vulnerability value is 0. However, these are ideal values because, up to now, it is difficult to determine the presence of a CVE and its exploitation with absolute certainty. Due to the previous assumptions (i.e. reachability issues, stripped binary, and imported libraries), the quantification must be expressed in probabilistic terms, on a scale between 0 and 1. The vulnerability factor is defined as the likelihood of the CVE being present in the binary. In this work, a qualitative measure is employed by assigning five levels:

- **CRITICAL**: the CVE is confirmed to be present and exploitable;
- **HIGH**: a vulnerable library is detected within the application, but the exploitability of the CVE cannot be confirmed. This may occur when the vulnerable API is not reachable or when the vulnerable function cannot be identified due to stripped binaries;
- **MEDIUM**: the same assumptions as in the HIGH level apply, but the vulnerable version cannot be determined because of stripped binaries. In this case, the library may still be vulnerable if developers were unaware of the function's dangerousness, for instance when the CVE was released after the application's publication. Applications in this category are

characterized by a difference between their release date and the CVE publication date of less than two years. According to Librarian [30], two years represent the typical timeframe developers take to apply a patch and mitigate the vulnerability after its disclosure. Hence, within this period, it is highly likely that the library is affected;

- **LOW**: the presence of the CVE cannot be established. A low level of risk is therefore assigned when the identified version and functions are not associated with any known CVE;
- **NONE**: no native library is detected, or the analyzed ELF files do not belong to the set of N products under consideration.

The goal of this study is to establish a qualitative value of the risk. To do so, the semi-quantitative product between threat and impact must be rescaled in a range between 0 and 100 as both of them have values between 0 and 10 into qualitative metrics, as detailed in Table 5.1. The values have been determined according to the CVSS 3.1 Qualitative Severity Rating Scale. Then, by applying the equation 5.1, the risk is evaluated according to the matrix in Table 5.1. In this way, a qualitative risk assessment score can be assigned to each CVE identified within the libraries of every APK. The purpose is to give a risk value to each library (and, consequently, to each app) and to provide swift alarms to developers. Once the developers are informed about the risks associated with the library or application, they can find a way to patch the vulnerability (e.g. upgrade the library to a non-vulnerable version, fix the library, etc.). To this aim, the highest CVE risk level identified within each library was assigned as the library's overall score, while the specific affected CVEs were stored in a log file. For instance, if five CVEs with a `LOW` level and one with a `MEDIUM` level are detected, the library is assigned a `MEDIUM` risk to raise a more effective alarm. An alternative approach could involve computing the average risk level to obtain the most representative score; however, in the example above, this would result in a `LOW` risk, thereby underestimating the actual threat, which is contrary to the intended purpose. The same weighted strategy was adopted for the attribution of the overall application risk level.

Vulnerability / Threat*Impact	None	Low	Medium	High	Critical
Critical	Medium	High	High	Critical	Critical
High	Medium	Medium	High	High	Critical
Medium	Low	Medium	Medium	High	High
Low	Low	Low	Medium	Medium	High

Qualitative value ranges: *Critical* (90–100), *High* (70–89), *Medium* (40–69), *Low* (0–39).

Table 5.1: Risk matrix to determine the qualitative and quantitative risk value.

This method has been applied to 100.000 APK from the Androzoo dataset, which aggregates millions of Android applications from various markets and time periods. Approximately 40% of these apps contained Native Code, amounting to 38.384 APKs. I focused the analysis on 15 high-impact, widely used products (e.g.`OpenCV`, `OpenSSL`, `FFmpeg`, `SQLite`, `Libpng`, `Libjpeg-turbo`, `Lua`, `Mono`, `Folly`, `Hermes`, and `React-Native`) chosen based on their prevalence and history of CVE. These

accounted for a significant portion of the Native Code in the dataset, allowing us to assign risk scores to roughly 55% of the libraries in APK. The results showed that 26% applications still contain native libraries with known vulnerabilities, some of which have been public for years. 55% of the apps fell into the `HIGH` or `MEDIUM` risk categories. Temporal trends indicated that older applications were more likely to contain unpatched vulnerabilities, but even recent releases sometimes carried high-risk components, suggesting delays or oversights in applying security patches. Market analysis revealed that the `Google Play Store`, while dominant in app distribution, had the largest number of vulnerable apps in absolute terms, reflecting both its size and the fact that its security checks focus primarily on malware detection rather than vulnerability scanning.

To validate the methodology, a comparison with the `Librarian` tool is made, shown in Figure 5.2. `Librarian` is one of the few existing systems designed for detecting Android Native Code vulnerabilities. Using `Librarian`'s dataset, I achieved equivalent detection accuracy, but the presented risk method offered the additional advantage of producing nuanced risk scores rather than binary yes/no outputs. When I analyzed updated versions of the same apps shown in Figure 5.3, I observed that 55% showed reduced risk, 10% remained unchanged, and 2% saw increased risk, with the remainder not assessable because they lacked the targeted products.

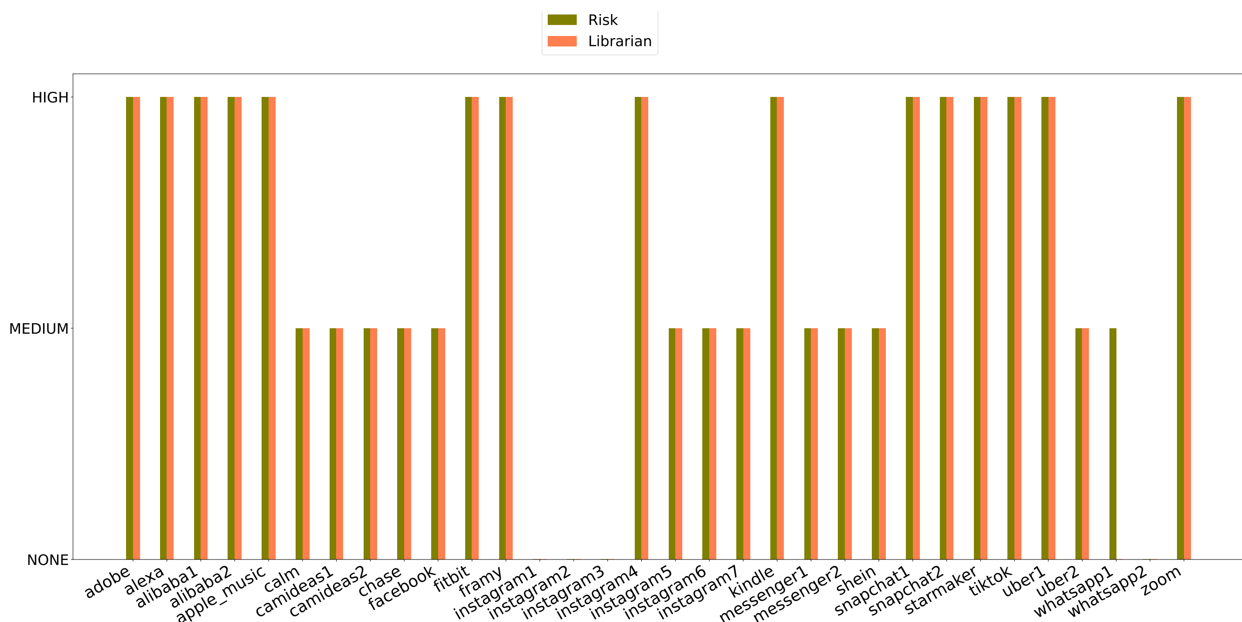

Figure 5.2: The histogram shows the risk level for the librarian apps computed according to the risk algorithm (left/olive green bar) and the librarian algorithm (right/coral bar). The two approaches are equivalent

These findings demonstrate that the risk approach can serve as a practical, scalable solution for early-stage vulnerability detection in Android Native Code [3]. By combining lightweight binary analysis, structured vulnerability intelligence, and a risk-based scoring model, it becomes possible to identify and prioritize vulnerabilities prior to release, thereby enhancing the security of the Android application ecosystem and contributing to stronger software supply chain resilience. Moreover, there is no complete ground truth study on Android Native Code vulnerabilities on which train specific AI-based algorithms for a faster detection. In this way, the risk study allows to give such knowledge

[3] `https://github.com/slsanna/Android_Native_Code_Vulnerability_Detection`

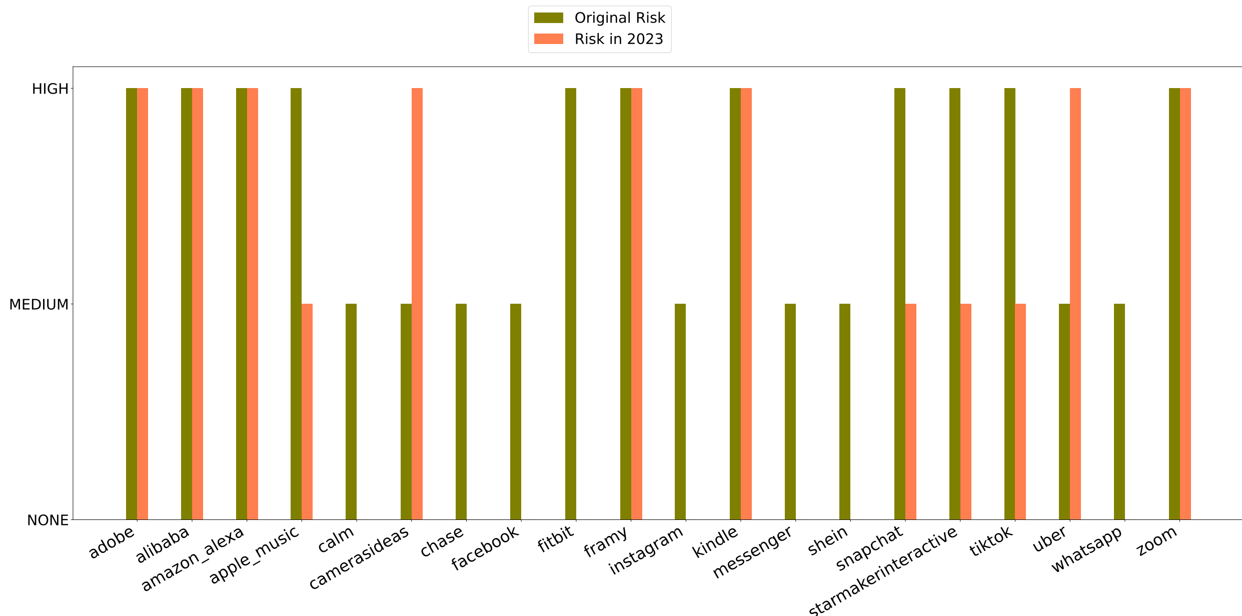

Figure 5.3: The histogram shows the risk level for the APK in the version used by Librarian (left/olive green bar) and the version on February 2023 (right/coral bar). The two approaches are equivalent

that, because of the manual validation, can be used as a base for automatic and fast detection, even for example taking inspiration from SPAM algorithms.

The presented work has been published in 2024 in Journal of Cybersecurity Oxford [179] and defines a score to prioritize vulnerabilities assessment.

5.2 Reaching Native Code Vulnerabilities

A limitation of the previous work is that the vulnerable functions are often detected in libraries without considering whether they are actually reachable from the application, i.e. whether control flow can propagate into them, asking ***How can the vulnerable function be called?*** In many cases, a vulnerable function may exist in a bundled library but never be invoked through the application's execution paths, which means the presence of the function alone does not imply exploitability. While dynamic approaches have been explored by Borzacchiello *et al.* [45, 46], which observe execution traces at runtime to confirm reachability, these techniques are limited by input coverage and execution environment. It is fundamental to explore the reachability of the vulnerable function because otherwise an AI-based algorithm will produce a significant amount of false positives and false negatives. Also for this part, I could not find any available public database on which to train the algorithm or a pre-trained network to be immediately applied to this analysis. For this reason, first I need to build the knowledge on the reachable path, i.e. the possible function graphs to be used to reach the vulnerability.

Current research on reachability in Android applications spans static, dynamic and hybrid techniques that each trade off soundness, precision and scalability. Static analysis (e.g. `FlowDroid` [37], `Amandroid` [209] and related call-graph/ICFG builders) attempt to infer reachable code and dataflows across Java and inter-component boundaries. Still they suffer from incomplete modeling of framework callbacks, JNI/native bindings and library interactions that produce spurious or missing edges.

Hybrid and symbolic/concolic approaches (e.g. concolic testing [182], `JDart` [128] and other symbolic engines) improve path-sensitivity and can synthesize inputs to exercise otherwise-elusive paths, yet they frequently face path explosion, framework modeling gaps and environment divergence that limit coverage at app scale. Dynamic, runtime approaches (execution tracing, UI fuzzers and automated-input generators such as `DroidBot` [124] or `Monkey` [93]) provide high-confidence confirmation of reachability for exercised paths but remain constrained by input coverage and the fidelity of the execution environment. Recent research has shifted toward analyzing native-side reachability, focusing on how vulnerabilities in native libraries can actually be triggered from an Android app's Java layer. These studies reconstruct JNI linkages to trace execution paths that cross from managed Java code into native C/C++ components. By modeling these interactions, tools such as `DroidReach` [45] apply specialized heuristics and targeted symbolic analysis to determine whether a vulnerability is genuinely reachable rather than merely present in the codebase. This selective, cross-layer reasoning significantly improves the accuracy of native vulnerability detection compared to traditional static analysis pipelines, which often overestimate exploitability by ignoring runtime linkages between Java and native layers. Taken together, the literature suggests that a practical, high-precision reachability solution for vulnerability triage will likely combine improved ICFG construction, targeted symbolic exploration for hard-to-model transitions, and lightweight dynamic confirmation to prune false positive, motivations that drive the static methodology I describe next.

In this section, I propose a static methodology that leverages the layered structure of Android APK to identify and confirm reachable vulnerabilities without requiring execution, hence in a faster and lightly computational way. Moreover, it is possible to extract all possible paths from any JNI function (source) to any vulnerable native function (sink), subsequently finding the shortest one for exploitation. The analysis begins by decompiling the application package APK to extract both the Java layer and the bundled native libraries. The decompiled Java code is inspected to identify dynamic library loading statements (e.g.`System.loadLibrary`) and native method declarations (e.g. public static native methods). These two elements are essential: the former reveals which shared objects are actually loaded at runtime, while the latter identifies the Java entry points into native code. Together, they allow us to filter out unused libraries that do not influence the application's behavior and to focus only on the subset of libraries and functions that are effectively linked to the app. The native library that interfaces with the JNI is thus isolated, as it provides the critical bridge between managed Java code and compiled C/C++ code. This step ensures that the subsequent analysis targets only those binaries that are exposed to the Java layer and therefore potentially exploitable.

Once the JNI boundary is established, the candidate libraries are analyzed using reverse-engineering techniques to construct two complementary representations. The first is a symbol and version identification. The binaries are scanned to enumerate functions that are cross-referenced within the code, thereby discarding synthetic or unused symbols typical of stripped binaries. In parallel, constant strings embedded in the binaries are mined to extract product identifiers and version numbers of third-party components. This information enables mapping each library instance to its specific upstream project release, such as `OpenSSL` build versions, `FFmpeg` releases, or `SQLite` distributions. Such mapping is fundamental, as many CVE are version-specific and only affect precise product releases.

The second representation is a directed call graph rooted in JNI entry points, i.e. the functions

associated to libraries to interconnect the Java code with the Native Libraries, allowing the runtime environment to automatically bind native methods to their corresponding Java declarations without explicit evocation. In fact, retrieving the JNI is fundamental for the reachability, otherwise the native function is never called. Functions whose names conform to JNI naming conventions (i.e.Java_<package>_<class>_<method>) are taken as root nodes, and a depth-limited traversal explores both incoming and outgoing function calls. Outgoing edges represent functions invoked by a given routine, while incoming edges capture callers that may propagate execution towards it. The resulting graph models the possible control-flow relationships from Java-level methods through the internal structure of the native libraries. This allows us to reason not only about which vulnerable functions exist, but also about whether execution paths exist that can lead from the application's external interface down to the vulnerable sink.

To detect known security flaws, the extracted version and function information are cross-referenced against a curated vulnerability knowledge base, i.e. the database constructed from the previous work. To recap, each entry in the database encodes semantic version constraints (e.g. "OpenSSL < 1.0.2g") together with function-level identifiers that are associated with a reported CVE. A vulnerability is flagged only when both conditions are met: the observed version falls within the vulnerable range and the vulnerable symbol is present in the binary. This dual check ensures precision by preventing false positives that would arise if, for example, a vulnerable function name exists in a patched library version, or if the library version is old but the specific vulnerable routine is absent. In fact, I made some experimentation in a restricted dataset from the previous study described in 5.1, the reachability reduced the risk of some application by decreasing the number of vulnerable functions (i.e. not callable) but increased the app exploitability by giving a certainty of the callable vulnerable function.

Finally, reachability analysis integrates the call graph with the vulnerability information to evaluate exploitability. For each vulnerable function, the system computes paths from Java-level entry points to the vulnerable routine. Graph traversal algorithms, such as shortest-path search, are applied to determine whether an application's code can feasibly trigger the vulnerable sink. If such a path exists, the vulnerability is considered reachable and therefore exploitable within the application context; if no such path exists, the flaw is considered inert, as the application lacks a mechanism to activate it. The results are summarized as a mapping from libraries to vulnerable functions, annotated with associated CVE identifiers and explicit call paths that demonstrate the linkage between managed Java code and unsafe native routines.

In this way, the methodology (shown in Figure 5.4) not only detects the presence of known vulnerable code within an application package but also establishes its practical exploitability. By combining decompilation, library version fingerprinting, function-level inspection, and graph-based reachability reasoning, it provides a more rigorous and actionable vulnerability assessment than approaches based solely on library fingerprinting or static pattern matching.

The native library can be modeled as a directed graph to trace paths from Java entry points to vulnerable functions. This methodology establishes whether flaws are truly reachable and exploitable. Building on this foundation, the next step is to leverage these vulnerability graphs for automation and learning, enabling AI to detect critical paths and provide interpretable explanations of exploitability. The systematic representation of native code as directed graphs naturally lends itself to ML approaches, as these structures capture both functional relationships and potential ex-

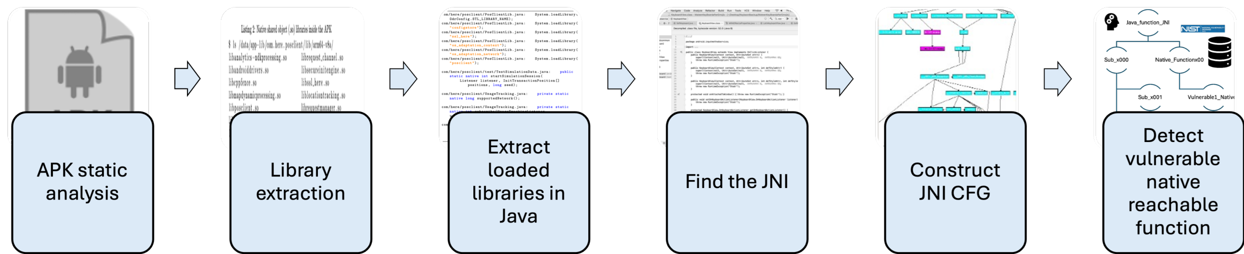

Figure 5.4: The picture shows the reachability pipeline where an APK is analyzed statically by extracting its library (second stage), its native libraries loaded in the Java code statically (third stage). Subsequently it is important to analyse all native libraries to find the JNI (fourth stage). Once detected, in the last stage the CFG can be computed from the JNI source function to the vulnerable native function (sink)

ploit paths. By collecting vulnerability graphs across a large number of applications, an algorithm can be trained encoding diverse vulnerability scenarios, ranging from trivial single-function exploits to deep multi-step call chains. ML models, particularly Graph Neural Networks (GNNs), graph embeddings, or tree-based classifiers, can then be trained to recognize structural patterns that correlate with exploitability. For instance, models can learn to distinguish between vulnerable nodes that are isolated and therefore unreachable, and those that lie on critical paths from application entry points. Training the AI to detect shortest paths is especially important, as it allows the system to automatically prioritize vulnerabilities that are not only present but also easily exploitable through minimal execution steps. In practice, this means that the model can flag high-risk vulnerabilities in new applications without requiring exhaustive manual graph traversal. Beyond simple detection, such AI systems could generalize to unknown vulnerabilities by identifying suspicious call structures or unusual dependency chains, thus complementing signature-based CVE matching with predictive capability. While AI models can greatly accelerate detection, their effectiveness depends on the ability of human analysts to trust and validate the predictions. To achieve this, the output of the vulnerability analysis can be visualized as a decision tree, where the root corresponds to a Java entry point, intermediate nodes represent successive function calls, and the leaves denote vulnerable sinks.

This tree-based representation has two key advantages: first, it mirrors the logical execution flow of the program, making it intuitive for developers and security analysts to follow; second, it aligns with the principles of xAI, as each branch provides a transparent rationale for why a given path is considered exploitable. Unlike opaque "black box" neural models, tree visualizations are intrinsically interpretable, offering insight into both the structural reasoning of the system and the specific evidence supporting a vulnerability claim. This interpretability not only increases confidence in the tool's output but also supports practical use cases such as vulnerability reporting, developer remediation, and forensic analysis. By combining AI-driven detection with interpretable visualization, the methodology ensures that results are both scalable and trustworthy, bridging the gap between automated analysis and human decision-making. We also claim that AI graph-based approaches can be used even to detect the unknown functions in stripped binaries and to better match the native compiled library with its relative product. In this way, a complete analysis of the native code is presented by automatically detecting with complete certainty the product, its vulnerable functions and how to call them in order to execute the exploit to inject in the RAM. To conclude, the vulnerable function detected from the risk-based approach can be called by

reconstructing its CFG from the Java function entry point, finding the JNI and following the CFG in the native library.

5.3 Exploiting Native Code Vulnerabilities

From the previous works, I assessed how to quickly determine a vulnerable function, than how to call it and now I investigate how to exploit the found reachable vulnerability, i.e. ***Which input triggers the vulnerabilities?*** Fuzzing [62, 129, 217] is an automated technique widely used to detect software vulnerabilities due to its simplicity, ease of deployment, and the large body of empirical evidence supporting its effectiveness. It works by repeatedly feeding specially crafted or malformed inputs to a program in an attempt to trigger vulnerabilities and cause crashes. This approach can uncover bugs and vulnerabilities that other testing techniques might miss, particularly by exploring different executable paths to improve code coverage. However, many fuzzers aim to cover as many paths as possible without prioritizing those that are more likely to be vulnerable without combining for example previous stages on vulnerability detection, assessment and reachability studies.

We remind that in the Android context, native code refers to compiled C/C++ code used in applications to interact with components such as the camera or microphone. These libraries may be written by the app developer or imported from third parties like `Libpng`, `OpenCV`, or `FFmpeg`. Vulnerabilities in native code, such as buffer overflows, format string issues, and heap overflows, can lead to memory corruption. If attackers identify such flaws, they can send malicious inputs to exploit the application, potentially accessing memory (RAM) contents or directly injecting code.

Fuzzing Android native libraries typically involves building or using a ready-made fuzzer compatible with the device's architecture (see section 3.1.1), selecting an APK containing at least one native library, and writing an exploitation script capable of interacting both with the native function and the Java code. Inputs can be random, mutated, or AI-generated and are subsequently fed through the fuzzer, monitoring crashes to identify inputs that trigger vulnerabilities. `AFL++` [2] in `Frida` mode is currently the only fuzzer designed for native code in Android applications. Setting up requires a rooted device or emulator, building `AFL++` and related tools for the specific Android architecture, and creating a fuzzing script that targets the vulnerability in the native code. This setup involves downloading the appropriate Android OS and NDK versions, obtaining the `Frida` framework, compiling `AFL++` with the necessary configurations, and extracting the target native library from the APK. Once compiled, the fuzzer, exploitation script, and libraries are deployed to the device, and `AFL++` runs by injecting its `Frida`-based tracing library into the target. I made practical tests revealing that running `AFL++ Frida` mode can be challenging, with differences in OS architecture and configuration often causing execution issues. Attempts on both physical devices (such as a `Samsung SM-A500FU`) and emulated environments (using different API level for x86 and x86_64 architecture) revealed limitations in stability and reproducibility. `AFL++` offers advantages like accuracy, the ability to build on a computer and run on a device, with also large-scale input generation. However, it is not scalable because it requires rebuilding tools for each Android version and ABI, as well as writing custom fuzzing scripts for every application.

Alternative fuzzing tools for Android include `Android-AFL` [1], a modified `AFL` for the Android kernel that supports ARM architecture, and `LibFuzzer` [8], which requires deep analyst knowledge of the target API and is not easily scalable. The state of the art presents a va-

riety of approaches targeting different aspects of Android fuzzing. `FANS` [127], `FuzzGen` [107], `DroidFuzzer` [224], `CraxDroid` [61], `CrashTranslator` [106], and `FASSFuzzer` [105] each focus on specific testing strategies, from fuzzing system services and inferring library APIs to exploring execution paths or rapidly detecting null pointer vulnerabilities. Other efforts like `IntelliDroid` [213] and `BinderCracker` [78] target malicious behavior detection and Binder interface fuzzing, while some extend fuzzing to domains like Android Automotive CAN interfaces or IoT device communication, as in `Diane` [161] and `Fuzz IoT` [60].

Despite these advances, there is still no published open-source tool specifically aimed at fuzzing Android applications for native code vulnerabilities. Current solutions often neglect function reachability and scalability. The proposed direction involves simplifying the setup of existing tools, ensuring fuzzing targets only reachable functions, and generalizing vulnerability detection for third-party native libraries using CVE data.

AI is rapidly emerging as a transformative element in fuzzing, with the potential to address long-standing limitations such as inefficient path exploration, the lack of prioritization for likely vulnerable code, and the heavy manual workload involved in crafting fuzzing scripts. Its strength lies in the ability to learn from extensive datasets of program behavior and vulnerabilities, and to generalize this knowledge in order to guide fuzzing toward the code regions most likely to yield exploitable flaws. One example is `NeuFuzz` [230], which integrates DL into a gray-box fuzzing framework to improve seed selection. While traditional fuzzers often aim to maximize code coverage without distinguishing between safe and risky paths, `NeuFuzz` trains a prediction model on labeled data representing vulnerable and non-vulnerable code paths. This allows it to learn the subtle patterns that tend to occur in vulnerability-prone regions. During fuzzing, it can then prioritize inputs that are statistically more likely to trigger a flaw, resulting in faster vulnerability discovery and a higher likelihood of uncovering deep-seated bugs that random or coverage-driven approaches might overlook. In practice, `NeuFuzz` has demonstrated its effectiveness on benchmarks such as on real-world applications, where it discovered dozens of previously unknown bugs, many of which were assigned CVE identifiers. `Learn-Fuzz` [91] represents another important advance. It uses NN-based statistical ML to automatically generate grammars for complex structured inputs. Many applications, such as PDF readers or multimedia processors, rely on rigidly defined input formats. Creating an input grammar manually for these cases is laborious and time-consuming. `Learn-Fuzz` bypasses this step by analyzing a set of valid inputs to infer their grammar rules. It then generates new inputs that conform to these rules, ensuring they are realistic enough to traverse meaningful code paths, while intentionally injecting controlled anomalies to trigger error-handling routines and expose unexpected behavior. The balance between generating structurally valid inputs and introducing well-placed malformations allows for maximum parser coverage without neglecting vulnerability discovery.

The integration of AI into fuzzing can extend beyond these examples into several key capabilities. One promising direction is the automatic detection of vulnerable paths. By extracting control flow graphs from both static and dynamic analysis, and training models such as Recurrent Neural Networks, LSTM, or GNN on secure and insecure examples, it becomes possible to classify each branch of execution according to its likelihood of containing a vulnerability. This enables the fuzzer to focus computational resources on the code most worth exploring. Another direction is the automated generation of exploitation scripts. Once a vulnerable function is identified, LLM or

evolutionary algorithms could be used to produce variants of attack payloads tailored to specific vulnerability types such as buffer overflows, format string vulnerabilities, or heap corruptions. This would allow fuzzing to evolve from merely finding crashes to actively producing proof-of-concept exploits. AI can also be used to test fuzzers themselves. In an adversarial setting, it is possible to create inputs designed to evade vulnerability detection systems, simulating the tactics an attacker might use. This type of testing would help harden AI-powered fuzzers against real-world evasion strategies. Additionally, Reinforcement Learning offers the opportunity to adapt the fuzzing process dynamically. Inputs that lead to the discovery of new code paths or that produce unusual execution states could be rewarded, allowing the fuzzer to refine its input generation strategy in real time, gradually allocating more effort to promising directions.

Despite these advantages, the application of AI to fuzzing faces significant challenges. A major obstacle is the absence of high-quality, representative training datasets that capture the diversity of both safe and vulnerable code in realistic settings. This is way, in this topic I focused on the pure fuzzing methodology instead of testing it with the use of AI, still there is no public available dataset. Without such datasets, models risk overfitting to narrow benchmarks and failing when applied to novel software. Performance overhead is another concern, i.e. integrating DL models into the fuzzing loop can slow down execution, particularly when inference is computationally expensive. AI techniques also struggle with multi-point-triggered vulnerabilities that require a specific sequence of conditions to be met, as well as with entirely new vulnerability types that differ significantly from those seen during training.

In the specific context of Android native code fuzzing, the integration of AI could lead to systems capable of automatically parsing APK files to extract both Java and native components, mapping their API interactions, and statistically prioritizing the exploration of high-risk paths using graph-based learning. Once a vulnerable path is identified, the same system could generate targeted, multi-stage exploitation scripts, execute them in a controlled environment, and adapt its strategies based on the results. Over time, such a framework could continually improve its performance by incorporating knowledge from newly discovered vulnerabilities, building toward a self-improving, fully automated vulnerability discovery pipeline. If successful, AI-powered fuzzing for Android native code could transform the field from a largely brute-force and trial-and-error process into a precise, adaptive, and scalable approach capable of analyzing thousands of applications and architectures with minimal human intervention. This would not only enhance the efficiency of vulnerability discovery but also significantly expand the scope of what is possible in automated software security testing. The answer to the main research question of this work is that fuzzing techniques will help in identifying the input triggering the vulnerable function but currently there is no mature methodology on Android devices allowing for fast and immediate tests.

To conclude, by answering to the main research question that lead to this Chapter, the content of the RAM can be accessed by exploiting Native Code vulnerabilities, i.e. detecting a vulnerable native function, reconstructing its CFG and finding the input triggering the vulnerability.

The last works (i.e. reachability and fuzzing) are not still published.

Approach	Research Question	Contributions	Results	Limitations
Risk	Is it possible to define a score to prioritize vulnerabilities?	Methodology to estimate risk; fast technique to detect vulnerabilities	55% of APKs have *HIGH–MEDIUM* risk; 12% of APKs have remained at high risk for years	Reachability of vulnerable functions is not assessed
Reachability	How can the vulnerable function be called?	Methodology to generate a CFG reaching the vulnerable function starting from Java entry points	Combination of paths from a Java function to call the vulnerable C/C++ function	No large-scale evaluation due to lack of an appropriate public dataset
Fuzzing	What input triggers the vulnerability?	Testbench to fuzz applications and trigger vulnerabilities	Define an AI-based fuzzing methodology for Android ecosystem	No large-scale evaluation due to lack of an appropriate public dataset

Table 5.2: Native Code Contributions

Chapter 6

Memory Acquisition in Android Devices

Once a vulnerability in the Native Code of an Android applications is detected, the calling graph is constructed and the triggering input has been identified, an attacker can have access to the main memory. In RAM data is in clear text, hence sensitive data can be steal or specific commands can be executed to run an attack. For this reason, in this chapter, I answer to the second main research question of this thesis ***How to acquire the RAM?***, by presenting two tools (i.e. `MoLIFE` and `AndroMemDump`) on how to acquire Android memory to collect evidence for Android malware detection. In particular, I focus on two methodologies of RAM acquisition, i.e. target process and physical RAM. Android memory forensics can be applied only if the device is rooted, in fact in the first section I introduce this problem and how it can be solved from the DF perspective of evidence preservation.

The chapter is based on the acquisition methodologies and tools for mobile forensics answering to *How to forensically acquire a non-rooted Android device?*. Recent efforts to catalog and critically assess digital forensic tools highlight a fragmented ecosystem with issues in availability, maintenance, and scientific reproducibility. Wu *et al.* [214] reviewed nearly 800 academic publications and identified 62 distinct tools across forensic subfields, but found that only about half were publicly available and actively supported, underscoring the need for centralized repositories, consistent documentation, and modular architectures to enhance tool validation and reuse. In the context of validating novel forensic methods, this underscores the importance of rigorous tool evaluation as part of methodological contributions. According to this, in this chapter I highlight the importance of acquisition tools.

6.1 Root Problem

Modern Android layered security (e.g. locked bootloaders, verified boot/dm-verity, SELinux, and hardware-backed encryption) prevents direct access to key forensic artifacts stored in protected or volatile areas [194]. As a result, investigators adopt a tiered strategy: prioritize non-destructive extractions, use temporary RAM-based or agent-assisted methods when possible, and reserve exploit or hardware-level access as a last resort or something not completely acquiring private system partitions or volatile data, leaving critical artifacts beyond reliable forensic reach. The main research question of this Chapter is ***How to forensically acquire an Android device without root permissions?***

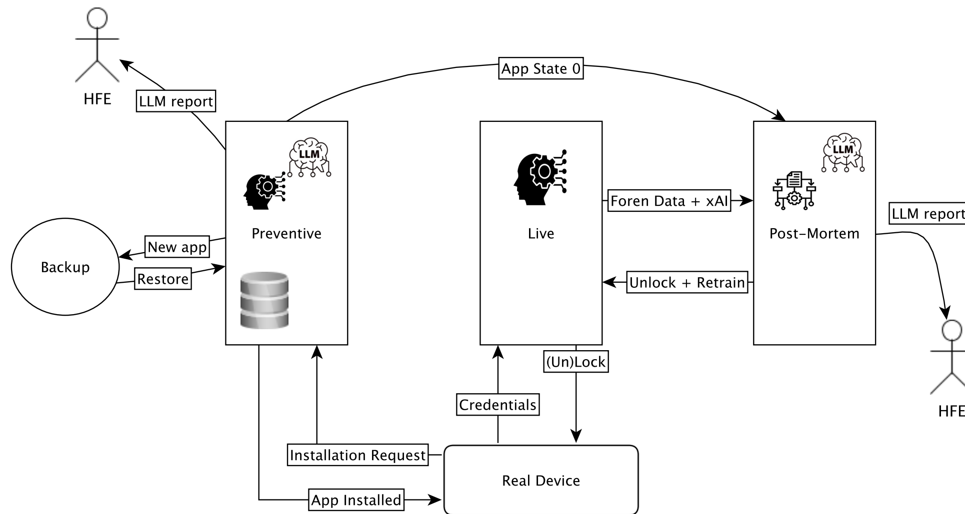

Figure 6.1: Methodology schema. The three stages *Preventive* with the clean backup, *Live*, and *Post-Mortem* are presented with the interaction between them. The Preventive stage receives the installation request from the real device, analyzes it, and installs the application in case of no anomalies, restores the clean state, and generates the report. In case of an anomaly, the Live stage receives the login credentials, locks the real device, and sends forensics data and xAI output to the post-mortem stage. The Post-Mortem stage receives the forensics incident data, the xAI, the app's original state, and, in case no anomaly is detected, sends an unlocking signal to the live stage to be redirected to the real device and generates the report with LLMs.

Addressing the rooting problem is threefold. First, *completeness*: without privileged access investigators can miss ephemeral or protected evidence that resolves intent, timing, or message contents (e.g. in-RAM decrypted images, session tokens, encryption keys). Second, *reproducibility and defensibility*: rooting changes device state in ways that are difficult to account for retrospectively in court, thereby undermining confidence in the original evidence. Third, *operational survivability*: exploit-based methods are inherently unstable and device-specific, yielding inconsistent results across the heterogeneous Android ecosystem and rapid patch cycles. Collectively, these pressures create an operational deadlock in which practitioners must choose between an incomplete but forensically "clean" extraction and a more complete yet legally contentious rooted acquisition. The remainder of this section motivates a method designed to break that binary trade-off.

The mobile Digital Twin reframes the problem with the `MoLIFE` workflow shown in Figure 6.1: instead of performing privileged operations on the physical evidence device, it creates an investigator-controlled, fully synchronized virtual replica that can be rooted and instrumented safely. A properly configured mDT mirrors inputs, network state and user interactions from the real phone across secure TCP/IP channels (for example, Wi-Fi debugging routed through an isolated VPN), enabling investigators to run the privileged, high-risk analyses on the twin while leaving the original device unaltered. This key design separates where privileged analysis happens from what is analyzed, preserving the evidentiary state of the device while still delivering root-level visibility.

The fundamental difference between a mDT and a traditional emulator lies in their operational philosophy and purpose. A Digital Twin (DT) is a virtual counterpart of a real physical entity, continuously synchronized with it to replicate its behavior, state, and operations in real time. It establishes a bidirectional communication channel (typically via TCP/IP) that allows live data exchange between the physical and digital domains. Unlike conventional emulators, which merely

simulate a device's hardware and software environment in isolation, the mDT maintains an active and dynamic link with the real device, processing the same inputs and reflecting the same outputs. This continuous synchronization enables analysts to monitor, experiment, and perform forensic analysis without directly altering the state of the evidentiary device. While emulators such as `Android Studio` or `Genymotion` offer static and independent environments primarily designed for development and testing, an mDT provides a synchronized and context-aware mirror of the physical device. This makes it particularly suitable for forensic and security applications, where maintaining evidentiary integrity and ensuring analytical reproducibility are paramount. Furthermore, the mDT can incorporate AI models for anomaly detection, behavioral correlation and predictive analysis. This does not have restrictions on computational resources as real devices, thereby extending the analytical and preventive capabilities of mobile forensics. In summary, whereas an emulator provides a controlled yet isolated sandbox, a mDT represents a live, intelligent, and evidentially faithful extension of the physical device, enhancing forensic analysis without compromising the original evidence. `MoLIFE` methodology demonstrates the mDT concept into three `NIST`-aligned stages with concrete tooling and governance, used for *preventive*, *live* and *post-mortem* analysis.

The *Preventive Forensics* stage acts as the first defensive layer of the `MoLIFE` framework, designed to prevent the installation of malicious or tampered APK before they interact with the real system. Its methodology follows a structured and automated workflow integrating hash verification, threat intelligence consultation, AI-based analysis, and explainable reporting.

The process begins when a user requests the installation of an APK. The system forwards the file to the mDT, where the first check is performed through hash matching. The APK hash is compared against a database containing verified legitimate applications and their previous versions to detect repackaged or unauthorized apps. Repacking is a widespread attack technique in which a legitimate APK is modified to insert malicious code or unauthorized functionalities while keeping the original appearance and behavior [189]. This practice is particularly common in cracked or premium-service applications, where attackers redistribute altered versions that appear genuine to users [13, 63]. Since the repacked APK typically preserves the same interface and functionality, it is indistinguishable to the end user but has a different cryptographic hash. For this reason, the preventive forensics stage uses exact and fuzzy hashing techniques (such as `ssdeep`, `feature-hash`, and `permhash`) to identify unauthorized packages. To complement this, code similarity analysis further examines structural differences between the submitted and legitimate versions, revealing inserted or modified code fragments [24].

The trusted hash database, stored on a dedicated server, maintains both current and historical versions of legitimate applications, ensuring that official updates are correctly recognized. To strengthen verification, CTI platforms such as `VirusTotal`, `MITRE CVE`, and `NIST NVD` are queried automatically using the computed hash. These external sources, enriched through LLM and NLP-based interpretation, provide contextual insights on known vulnerabilities and potential threats. If the hash and CTI checks are passed, the APK undergoes AI-driven analysis within the mDT using a pre-trained model designed to detect suspicious behaviors or vulnerabilities. All analyses are conducted inside isolated virtual environments to maintain forensic integrity, and after each run, the virtual machine is restored from a clean backup to eliminate any residual traces or persistence mechanisms. The outcome of the analysis determines the system's response: if no threat is detected, the APK is approved for deployment to the Live Forensics stage and subsequently to

the physical device. Otherwise, installation is blocked, and an automatically generated LLM-based report summarizes the identified risks to assist human analysts. xAI components then interpret the model's decision, highlighting the features that influenced the classification to ensure transparency and accountability in automated assessments.

According to the NIST standard, this stage follows all the phases: the (i) *collection* phase to identify data from the running application; the *(ii) examination* because it extracts potentially risky features from the application; regarding the *(iii) analysis* phase, the system automatically interprets and takes a decision on the extracted data with a conjunction analysis from external CTI tools and in case of a detected suspicious behaviors, *(iv) report* it to the human analyst.

Sometimes malware reveals its malicious behavior after an elapsed time or event, or a vulnerability is exploited when someone discovers it or by specific user input [83]. Moreover, the preventive analysis is fast due to the short time a APK can be kept in "quarantine" because the user needs it, especially in critical scenarios. For this reason, the preventive forensics stage alone is not sufficient. Accordingly, `MoLIFE` needs *Live Forensics* analysis to continuously monitor the APK and its effects on the device at runtime and during standard execution. The live forensics stage supervises and collects forensics data in case of future possible DF analysis with a more forensically oriented algorithm. For example, the full RAM content can be analyzed to check the effects of a specific target APK in the whole system at runtime. In addition, the target APK can be inspected with instrumentation tools (e.g.`Frida`) to check specific behaviors at run-time. The network connections and communications can be inspected to check which IPs and domains the APK contacts and retrieve the communication payload. As the mDT has super-user privileges, specific directories can be monitored to check if uncommon and potentially dangerous files are downloaded, created, or accessed by the APK at runtime and stored in the device. It should be noted that this simulation technology is potentially dangerous, as the mDT includes intellectual property and personal data, whose disclosure could cause significant consequences and irreparable damages [119, 195]. In this scenario, the mDT can check if specific, unauthorized, illegitimate, and uncommon data is read and exploited by the APK outside of its classic, standard, and declared behavior obtained from the previous preventive forensics stage investigation. The forensics-based AI model used must be robust to adversarial attacks to avoid an attacker using anti-forensics techniques [112, 142] and steganography techniques [56, 189] to hide behaviors and bypass detection. Even in this stage, the AI algorithm includes xAI-based techniques to find the features that have the highest influence in classifying the behavior as a security threat, and the related reason. This is important for the next post-mortem analysis stage to reconstruct what happened. In case of a cyber attack, the APK is blocked first on the real device to prevent more damage, avoiding desynchronization, and, lately, on the mDT. Subsequently, the collected data are sent to the post-mortem forensics to reconstruct what happened automatically.

This stage follows the NIST standard in the *(i) collection* phase when it identifies the forensics data to which to pay attention during the APK execution; the *(ii) examination* phase by extracting information and features from the identified forensics data; the *(iii) analysis* to interpret the forensics data under attention. This phase does not have a traditional *(iv) reporting* phase because no final report is generated for the human analyst, but in some sense, sending the features that determined the threat classification and the collected data to the next stage could be considered a sort of report as the information is described.

Post-Mortem Forensics

The *Post-Mortem* stage investigates with fuzzing techniques whether a detected threat originated from user input [183] or from hidden APK behavior by reproducing the application's state at the time of the incident. It consumes three primary inputs: the original APK snapshot from the preventive stage (i.e. state-0), the Live Forensics incident data (i.e. forensic artefacts, runtime state, xAI features that triggered the alarm), and CTI reports. Its goal is to reconstruct the triggering sequence and produce evidence to decide if the AI in the live-stage requires retraining or if the incident represents a genuine attack. The mDT replays the APK from state-0 while iteratively supplying crafted inputs until it reaches the same forensic state observed during the incident. This guided input generation (i.e. white-box because the APK structure is known) can be assisted by or AI models that synthesize wide-coverage, semantically valid test cases informed by the preventive-stage analysis [117, 213, 217]. Each execution is monitored for crashes, anomalies, and deviations; inputs that reproduce the incident state are logged and returned as evidence. If the fuzzing algorithm reproduces the incident, the system identifies which input or event triggered the behavior and generates an LLM-assisted report for human analysts; execution remains blocked until reviewed. If the fuzzer cannot reproduce the incident, this suggests a modeling error or advanced anti-forensics techniques; the system notifies operators to retrain the live-stage AI and, after retraining and validation, resumes execution first on the mDT and then on the real device. As in the preventive stage, CTI data are integrated through to enrich context and improve report clarity without affecting the algorithm's decisions.

This last stage follows the NIST standard in the *(i) collection* because it receives the data from the preventive forensics stage (i.e. the original APK and its analysis). The live forensics system (i.e. the state of the APK at the moment of the incident, the xAI features) and from external CTI tools. The NIST *(ii) examination* phase is used to interpret all the received data, the original APK, and the incident state. The *(iii) analysis* phase corresponds to the check at each step of what happened and how the incident occurred by using the fuzzing techniques. When the incident state is reached, or after many analyses, no threat is detected, the system gives in output a *(iv) report* of the findings to the human forensics analyst.

AI is the analytical backbone of the `MoLIFE` methodology, operating as an interpretable, auditable decision layer across all three forensic stages. In the preventive phase, lightweight classifiers and code-embedding models analyze static and dynamic features of applications (e.g. permissions, API usage, native libraries, and repack signatures) comparing them with CTI feeds to detect anomalies before installation. During the live stage, streaming anomaly detectors and sequence models continuously monitor behavioral signals such as network flows, system calls, and memory features captured from the mDT, flagging deviations from baseline activity. Each alert is accompanied by xAI algorithms, clarifying which features triggered suspicion, ensuring decisions remain transparent and defensible. In the post-mortem phase, AI guides deterministic replay and fuzzing: and reinforcement-based fuzzers generate realistic input sequences to reproduce malicious behaviors observed in memory or network traces, providing a data-driven reconstruction of events. Throughout these stages, `MoLIFE` enforces model governance—version control, model cards, continuous evaluation, and adversarial training to ensure reproducibility and resilience against evasion. AI is therefore not a black box but a forensically accountable assistant: it accelerates triage, detects hidden behaviors in real time, and reconstructs incidents with mathematical and legal transparency, while

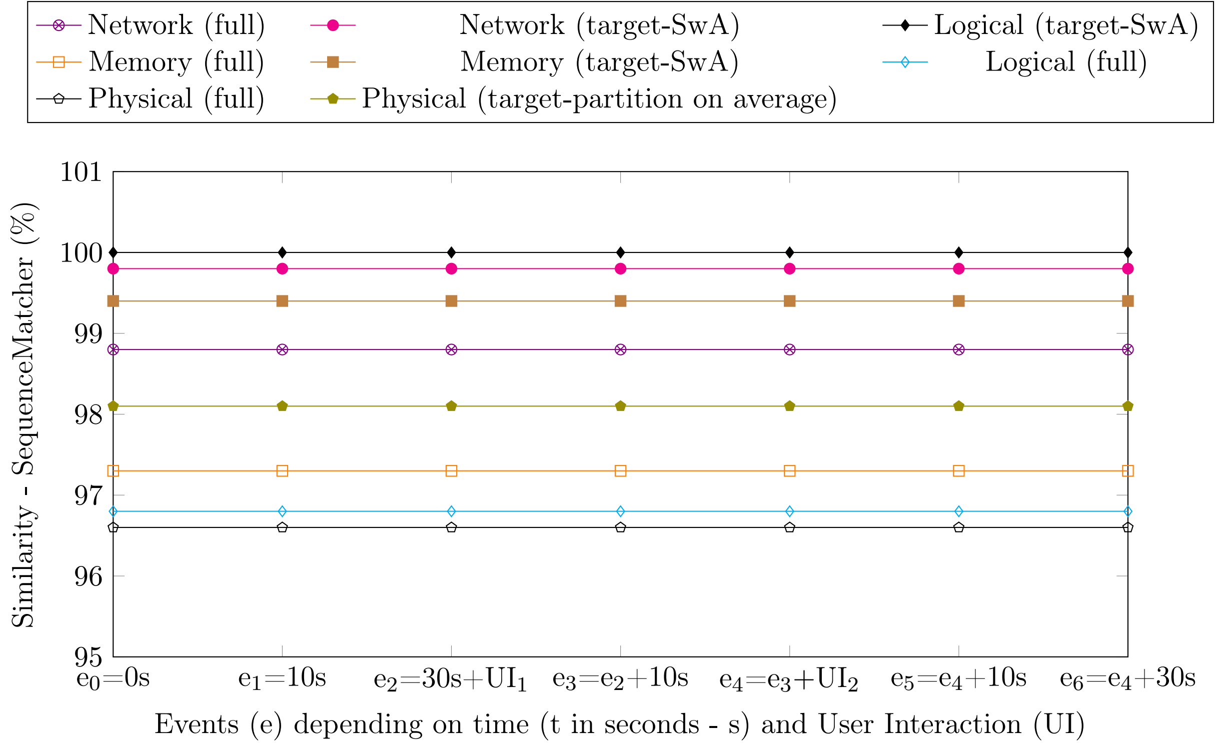

Figure 6.2: Acquisition percentages over time (t) and user interaction (UI). The graph shows, for the 8 different forensics acquisition methodologies, the similarities between the acquired analyzed data between the mDT and the real device under the same running condition (same architecture, OS, running application and version, running time, and user inputs). Similarity has been computed with the *Ratcliff/Obershelp* algorithm

keeping all decisions traceable, explainable, and under human oversight.

The `MoLIFE` framework integrates communication systems, AI algorithms, and high-performance computing for the mDT, all aligned with cybersecurity principles, i.e. confidentiality, integrity, availability (CIA), resilience, and non-repudiation. Quantum computing remains a future enhancement [115] for improving processing speed, communication security, and the overall performance of AI-driven forensic analysis within `MoLIFE`. Blockchain ensures data integrity, authenticity, and traceability through its decentralized, immutable ledger, while access control, encryption, and logging protect other components. Availability is reinforced by distributed and redundant infrastructures, confidentiality by encryption and anonymization, and resilience by robust AI models and hardened systems. Non-repudiation is provided by blockchain's auditable records and communication traceability.

In `MoLIFE` Android validation experiments, a systematic comparison between acquisitions performed on a rooted physical device and its synchronized mDT under identical operational conditions has been conducted. Both environments executed the same set of widely used messaging applications (i.e. `WhatsApp, Telegra, Facebook Messenger, Signal`) running on equivalent Android versions and with the same user interactions and execution timelines. The physical device used for benchmarking was a `Samsung A33`, selected for its balance between performance, accessibility, and compatibility with controlled rooting procedures, while the mDT was instantiated as a fully rooted virtual instance configured to replicate the same hardware profile, OS build, and network topology. Our comparison focused on four principal acquisition domains: logical extractions from

Figure 6.3: Image retrieved with carving methodology from the memory dump of the Telegram application in the mDT but the image visualized in the real device. The context of the image can be clearly and unequivocally seen

`/data/data` performed via ADB and forensic imaging tools, full physical images of system partitions obtained with dd and mounted disk exports, network packet captures collected using `tcpdump` and `pcapdroid` during live communication sessions, and complete volatile memory dumps acquired through `LiMe` and parsed with `Volatility` or acquired with `Fridump` focusing on the target app and parsed with strings or carving tools.

Across all evaluated dimensions, the results demonstrated a consistently high degree of alignment between the mDT and the physical device. For target-application logical data (e.g. SQLite databases, configuration files, and cached multimedia) the extracted artifacts were virtually identical, matching in filenames, metadata, and file hashes, with Ratcliff/Obershelp [1] sequence similarity reaching 100% within application-specific directories. Whole-system physical and memory images showed differing raw hashes due to inherent architectural variations between vendor-specific partitions and the emulator environment, for example, Samsung proprietary components that cannot be replicated or `Android Studio` `Genymotion` utilities absent on real devices. Nevertheless, sequence-based comparison revealed over 95% content similarity across physical, logical, and RAM artifacts. Network captures likewise displayed near-total correspondence in structure and payload, confirming that the mDT accurately mirrored runtime behavior, API interactions, and encrypted communication flows. The few observed discrepancies (i.e. emulation gap) were limited to non-probative system areas, mainly involving missing vendor partitions (e.g. Samsung's /efs or Knox services) and emulator control processes (e.g. `Genymotion` or `AVD` management agents). None affected user-level artifacts or application-layer evidence. All divergences were recorded in the final report, accompanied by similarity ratios and cryptographic hashes for transparency and reproducibility. Overall, the findings confirm that a properly synchronized and instrumented mDT can replicate the evidentiary substance of a rooted physical device with over 95% fidelity across all acquisition categories, while preserving the integrity of the original system. This demonstrates that mDT can function as forensically reliable analytical mirrors, capable of reconstructing both persistent and volatile artifacts when emulator limitations are properly documented and methodologically controlled.

The most significant outcome emerged from the memory analysis. Through full RAM inspection on the mDT, it is possible to recovered successfully transient artifacts (e.g. decrypted strings, in-memory session tokens, and ephemeral media fragments) that are typically lost during reboot or rendered inaccessible after rooting a physical device. In particular, I was able to recover `Telegram`

[1] `https://docs.python.org/3/library/difflib.html#module-difflib`

one-shot images and other short-lived media directly from the mDT volatile memory, verifying that these artifacts persist long enough to be extracted in a forensically controlled virtual environment as shown in Figure 6.3.

The experiments specifically show recoverable ephemeral artifacts: one-shot images displayed in Telegram were present in clear within RAM and retrievable via full memory dumps and carving tools (e.g. `binwalk`). There are important caveats: `Signal` does not permit cloud/import of message history to another device, and some secret/TE-bound chats remain device-exclusive, so the twin cannot reproduce hardware-protected primitives; where vendor-bound services or Secure Element data matter, hybrid vendor/export or hardware methods remain necessary. When the real device is unrooted but the owner cooperates (`Google Backup` enabled), the mDT can act as a synced "new device" to obtain the same app data and produce identical file hashes; when the owner does not cooperate, installing the same app on a rooted mDT still allows exploration of storage locations and a risk assessment of what root-level acquisition would reveal. Overall, the experiments show that a well-synchronized, rooted mDT can reproduce persistent and many volatile evidentiary artifacts with high fidelity while preserving the original device's integrity, despite provided emulation gaps (vendor partitions, TEE/SE behavior, well documented).

The Android case study validates the central `MoLIFE` claim: a mDT can restore broad analytic completeness without touching the original device, reproducing network, logical and many RAM artifacts with very high content similarity. Yet the approach has explicit limits, i.e. vendor services, Secure Element/TEE-bound secrets, and anti-emulation or root-detection behaviours cannot always be reproduced. These must be reported as emulation gaps in any forensic output. Finally, because mDT infrastructures are dual-use, they pose real anti-forensics and attacker-training risks: adversaries could use identical twin setups to probe defenses, tune evasion, or study how to erase traces while keeping a device superficially intact. `MoLIFE` therefore prescribes strict hardening, limited distribution of twin toolchains, and immutable auditing as mitigation steps; combined with hybrid fallbacks (vendor exports, hardware reads when authorized) and explicit similarity metrics (e.g. Ratcliff/Obershelp scores), the mDT becomes a pragmatic, auditable instrument that meaningfully reduces the need to root evidence while acknowledging the remaining technical, legal and adversarial boundaries.

The presented work is under review in an important Journal, a pre-print version is available online [174] and describes how to acquire an Android device without root permissions by adopting the concept of Digital Twin to mobile devices (mobile Digital Twin - mDT), having a complete synchronous copy of the target mobile device but with super user privileges and acquiring data from the mDT as if it was the physical device.

6.2 Memory Forensics Acquisition

Memory Forensics is the extraction and analysis of the volatile memory in a system by adopting different tools and techniques 3.3. It is fundamental as data is temporarily stored in clear text, without any encryption, and different important data can be retrieved both from a malware analysis perspective and from a data exfiltration or DF investigation. Despite its importance, memory forensics remains an underexplored area that requires further study, especially in mobile and particularly Android. First of all, it is important to release a clear methodology acquisition device-independent,

i.e. ***How to acquire the Android RAM?*** Secondly, efficient and robust analysis techniques must be developed. AI techniques can greatly streamline and enhance volatile memory analysis by automating complex detection and classification tasks. DL and NLP-based systems can recognize meaningful data patterns within raw memory (e.g. chat fragments, credentials, or cryptographic keys) by learning from semantic and structural contexts rather than fixed signatures. ML models can also classify processes and memory objects, distinguishing between code and data regions, and identifying injected or anomalous processes that deviate from normal behavior. Moreover, AI-driven anomaly detection enables the identification of subtle deviations, including abnormal API sequences or obfuscated code, which often indicate specific, stealthy or anti-analysis (e.g. obfuscated, steganographed, adversarial) malware. Finally, AI methods improve speed and scalability, processing large memory dumps in parallel and adapting to new operating system versions without requiring manual parser updates.

6.2.1 Overview on Android Memory Acquisition

Modern Android memory acquisition has evolved from full-RAM dumping tools like `LiME`, `fmem` and `VolMemDroid` [114], which required kernel access, toward lightweight and selective frameworks such as `Fridump`, `JIT-MF` [40], and `VEDRANDO` [41]. These newer methods enable targeted, real-time, and often non-root extraction of volatile artifacts (e.g. decrypted payloads and cryptographic keys) complementing system-level tools such as `Volatility` and `DroidScraper` [28]. Overall, research now prioritizes runtime, non-intrusive, and context-aware acquisition over traditional bulk memory imaging. Despite these advancements, current memory acquisition approaches still face several limitations. Many selective or non-root frameworks rely on instrumentation hooks that can alter application behavior or leave detectable traces, potentially affecting evidentiary integrity [43]. Others suffer from limited coverage, capturing only user-space memory and missing critical kernel or system-level artifacts [28, 40]. Performance overhead and real-time instability remain issues for tools employing dynamic instrumentation or virtualization [41, 42]. Furthermore, compatibility is often restricted to specific Android versions or architectures, limiting reproducibility across diverse devices. Finally, few studies provide formal validation or standardization of their acquisition methods, leaving questions about forensic soundness and cross-tool consistency.

For this reason, in this first section we provide a tool `AndroMemDump` [2] with a comprehensive methodology for acquiring the complete RAM (i.e. full RAM acquisition) of an Android device and the RAM portion allocated by a target process (i.e. process allocated RAM acquisition). The first one is useful to study the effects of an application (i.e. malware) in the whole system, while the second one is better employed for malware classification. Both methodologies have been developed and validated for real and emulated devices. Even if relying on two popular tools (`Lime` and `Fridump`) for memory acquisition, I noticed that there is no public and reliable working methodology and pipeline for Android memory acquisition. Moreover, there is no public dataset and for this reason I will release both the methodology acquisition and the acquired dataset.

6.2.2 AndroMemDump - Full RAM

`LiME` (Linux Memory Extractor) is an open-source kernel module designed for the DF acquisition of

[2] `https://github.com/slsanna/AndroMemDump`

volatile memory from Linux- and Android-based systems. Operating entirely within kernel space, `LiME` reads physical page frames directly and exports the captured memory either to local storage or through a TCP stream to a remote host. Its design enables investigators to obtain a complete, system-wide snapshot encompassing both user-space and kernel-space artifacts. The streaming option minimizes writes to the target's local storage, thereby reducing the risk of overwriting volatile data and simplifying integration into automated pipelines. The resulting memory image is contiguous and immediately compatible with established forensic frameworks such as `Volatility`, facilitating structured post-analysis. `LiME` remains the standard for full-memory acquisition in Android research because it offers a balance of forensic completeness, reproducibility, and practicality. Being open-source and architecture-agnostic, it can be audited, adapted, and recompiled for diverse kernel versions and hardware architectures. Moreover, `LiME`'s operational transparency, reproducible build process, and broad community validation make it suitable for academic and evidential use. Unlike hardware-assisted methods such as JTAG or DMA-based acquisition (which require physical access, specialized equipment, and entail a high risk of data corruption), `LiME` operates entirely in software, can be scripted remotely, and is therefore ideal for controlled research environments. Its principal limitation, however, lies in its requirement for a compatible kernel with module-loading capability and root privileges. On devices where module insertion is disabled, signed, or restricted, hardware or vendor-level extraction techniques (e.g. TrustZone debuggers, hypervisor snapshots, or DMA-based tools) become necessary alternatives.

The proposed methodology called `AndroMemDump - Full RAM`, illustrated in Figure 6.4, establishes a reproducible and forensically sound framework for full-RAM acquisition on Android platforms and answers to the main research question ***How to acquire the full RAM of an Android device?*** `AndroMemDump - Full RAM` is designed to capture an exhaustive snapshot of volatile memory (both user and kernel space) under strictly controlled experimental conditions. To eliminate host-side variability, all kernel and `LiME` build operations are executed inside a version-controlled Linux VM. This controlled build environment mitigates common cross-host inconsistencies such as missing libraries, header mismatches, or toolchain spurious artifacts, ensuring identical build outputs across experiments.

Since the Linux kernel enforces strict isolation between user and kernel space, privileged acquisition must occur within kernel context. Consumer and emulator kernels often disable dynamic module loading or enforce signed modules, preventing third-party forensic instrumentation. Consequently, kernel recompilation is a necessary precondition for memory acquisition. A custom kernel is therefore built from the official Android source tree, with module support explicitly enabled and signature verification disabled. This controlled recompilation preserves kernel integrity while enabling forensic module insertion without altering the target's runtime semantics. Rebuilding the kernel from source also guarantees ABI transparency. The latter is an essential property, as every Android version, device build, and architecture defines unique symbol tables and structure layouts that must match the expectations of any in-kernel acquisition tool.

The preparatory phase begins with target characterization: the analyst identifies the device architecture, Android API level, vendor or ROM build identifier, and kernel revision string (commit tag from the Android or vendor tree). This mapping step is essential, as even minor vendor patches or build-time macros can alter structure offsets or symbol exports. The device's build metadata is used to select the exact upstream repository and commit that generated the running image. When

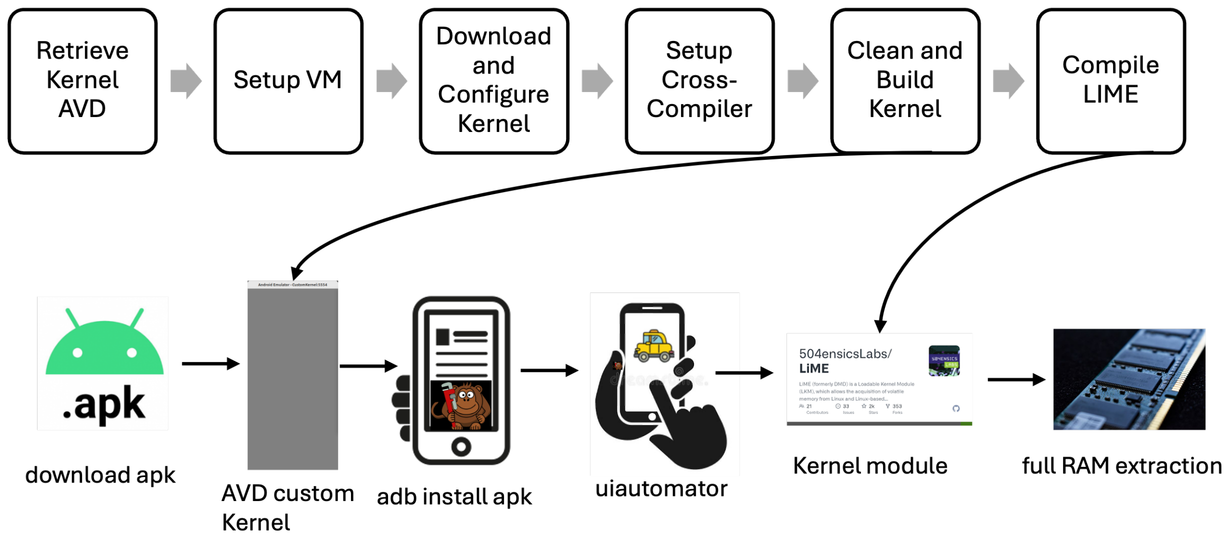

Figure 6.4: The picture shows the full RAM acquisition process. Above is reported the methodology for kernel recompilation and below the acquisition. Starting from the installation of an APK in a device or emulator with recompiled kernel, when the app starts executing, the compiled Kernel module is loaded and the full RAM is dumped

the kernel tag is not explicitly provided, repository and branch selection follow a deterministic heuristic prioritizing: *(i)* explicit commit identifiers from the kernel string, *(ii)* vendor branch by build fingerprint, *(iii)* kernel version and API-level proximity, and *(iv)* vendor backports where applicable. Each target runtime is thereby associated with a single, uniquely identified kernel source tree archived for reproducibility. From experiments, we noticed that general-purpose LLM (e.g. chatGPT) can quickly help in this identification process. Moreover, further matches or even AI-guided identification algorithms can produce a fast database for already-compiled kernels, even following the methodology reported here.

Once the correct kernel source is selected, the corresponding `.config` file is retrieved (i.e. from `/proc/config.gz`) and adjusted to enable loadable modules and disable signature verification mechanisms relevant to the controlled environment. Because kernel configuration directly influences structure layouts and symbol visibility, the final `.config` used for compilation is preserved as a primary experimental artifact. Kernel recompilation employs a locked, versioned cross-toolchain matching the original build (usually a specific `GCC` or `Clang` release); its exact binary versions, paths, and non-default flags are recorded in a manifest. The build process produces the kernel image, `vmlinux`, `System.map`, and exported headers required for out-of-tree module compilation. All outputs are archived to guarantee later verification and `Volatility` profile generation.

Algorithm 1 Build & Use Custom Android Kernel, LiME, and Volatility Profiles (Vol2/Vol3)

1: **Acquire Target .config from AVD**: `adb pull /proc/config.gz . && gunzip config.gz`
2: **Prepare Build Environment (Ubuntu VM)**: `mkdir ws && cd ws`
3: **Fetch & Configure Target Android Kernel Source**:
4: `git clone https://android.googlesource.com/kernel/$KFLAVOR && cd $KFLAVOR`
5: `git checkout $KBRANCH ; cp ../config .config ; make menuconfig` (ensure *module* support)
6: **Install/Export Cross-Compilation Toolchain**:
7: `export ARCH=$KARCH SUBARCH=$KARCH CROSS_COMPILE=$TRIPLE-`
8: **Build Kernel Artifacts**: `make mrproper && make modules_prepare && make -jN`
9: Preserve `vmlinux`, boot image (`Image`/`bzImage`), and `System.map`
10: **Compile LiME Out-of-Tree Module**:
11: `git clone https://github.com/504ensicsLabs/LiME && cd LiME/src`
12: `make KDIR=/path/to/$KFLAVOR` → `lime.ko`
13: **Boot Emulator with Custom Kernel Image**:
14: `emulator -avd $AVD -kernel /path/to/Image -no-snapshot`
15: Verify: `adb shell uname -a`
16: **Capture Memory with LiME on Device**:
17: `adb push lime.ko /sdcard/`
18: `adb shell su -c ’insmod /sdcard/lime.ko "path=/sdcard/ram.lime format=lime"’`
19: **Create Volatility Profile / Symbols (choose one line below based on version):**
20: *Volatility 2.x (profile zip)*:
21: In `volatility/tools/linux`: `make` → `module.dwarf`
22: `zip -j Linux_KERNEL-VERSION.zip module.dwarf /path/to/System.map`
23: *Volatility 3 (JSON symbols)*:
24: `dwarf2json linux -elf /path/to/vmlinux -system-map /path/to/System.map > linux_KERNEL-VERSION.json`
25: **Run Volatility Analysis on RAM Dump**:
26: *Vol2*: `python vol.py -f ram.lime -profile=Linux-KERNEL-VERSION linux_pstree`
27: *Vol3*: `python3 vol.py -f ram.lime -single-location=file:linux_KERNEL-VERSION.json linux.pstree`

Once the kernel and `LiME` module are verified as ABI-compatible, the target system is booted using the newly compiled image. The module is transferred to the device (`adb push`) and inserted through a privileged shell. The acquisition output is saved either to local storage (e.g. sdcard) or streamed over TCP to a remote collector, depending on the configured format. Each dump is cryptographically hashed and accompanied by environment metadata (kernel version, module commit, toolchain manifest) to ensure forensic reproducibility. The resulting image is subsequently processed with `Volatility`, using the generated symbol tables or JSON profiles to reconstruct process trees, memory maps, and kernel object states. Tests were conducted only on Linux Goldfish which is the most used kernel version adopted by different Android API and vendors and by other academic works like `VolMemDroid`. By investigating the different Android kernel versions and their specific requirements, and according to official documentation and reference sources, the presented approach can be adapted to all of them. While proper kernel compilation tailored to each target version is required, the overall workflow remains consistent. The proposed memory acquisition methodology is broadly applicable across Android kernel versions, although its reliability varies

with kernel evolution. On older kernels such as 3.4, 3.10, and 3.18, the approach operates smoothly since module loading is fully supported and signature verification is typically absent (i.e. up to Android API 26, which, although dated, still retains measurable worldwide use). Kernels from the 4.x series (up to around 4.14) remain largely compatible, though certain builds may necessitate minor adjustments, such as disabling module signature enforcement (i.e. up to Android API 29). Beginning with kernel 4.19 and especially with the introduction of Generic Kernel Image (GKI) versions like 5.10 and beyond (i.e. up to current Android API 36), the methodology encounters increasing restrictions, as modern Android builds often enforce strict module signing and may disable dynamic module loading altogether. In such cases, successful deployment generally requires setting SELinux to permissive mode. Due to the unavailability of physical devices, tests on on newer GKI-based systems were unfeasible; however, based on the documented kernel behavior, the procedure should remain technically applicable. As the method is not officially supported, I disclaim responsibility for potential device instability or bricking.

`LiME` provides complete visibility over system-wide volatile memory, including user-space processes, kernel objects, caches, and inter-process communication buffers. Its main strengths are: *(i) forensic completeness*, since it acquires the entire physical memory; *(ii) reproducibility*, as it can be deterministically built from source and verified against the running kernel; and *(iii)* practical *deployability*, because it integrates seamlessly into existing forensic toolchains such as `Volatility`.

Compared to user-space acquisition tools like `Fridump`, `LiME` operates at a lower level, yielding a holistic system image rather than a process-scoped snapshot. This enables cross-process correlation, kernel-level artifact recovery, and deep system reconstruction. However, these advantages come at the cost of higher intrusiveness and privilege requirements. Module insertion modifies the kernel's runtime state, however slightly, and can be infeasible on modern locked or production devices that enforce verified boot or kernel signature checks. Furthermore, the resulting memory image is considerably larger, imposing higher storage and analysis overhead. In contrast, `Fridump` provides finer granularity, lower privilege requirements, and reduced data volume, making it more suitable for targeted, hypothesis-driven investigations. Yet `Fridump` cannot access kernel structures or inter-process artifacts, not substituting full acquisition when system context is required.

In summary, `LiME` represents the most comprehensive software-based approach for volatile memory imaging on Android, offering full system visibility and strong reproducibility guarantees. It is best employed in laboratory or research settings where kernel modification is permissible and full coverage is essential. Conversely, `Fridump` remains preferable for application-level studies prioritising precision, ease of deployment, and minimal invasiveness. The two methodologies are complementary: `Fridump` yields high-resolution process-specific insight, whereas `LiME` provides the global forensic context necessary for complete reconstruction of system state.

This work with the text encoding analysis is under review in a conference and for double blind review could not specify which one. However, `AndroMemDump - Full RAM` presents a clear and unique methodology for the full RAM acquisition of Android devices.

6.2.3 AndroMemDump - Process RAM

`Frida` is a user-space dynamic instrumentation framework designed to inject lightweight agents into running processes and to expose a scriptable interface for introspection and runtime manipulation. It operates either through a privileged daemon (`frida-server`) on rooted devices or as an

embedded shared library (frida-gadget) integrated within an application package for non-rooted environments. Once injected, Frida provides APIs such as memory.scan() and memory.read*() that enable enumeration and reading of a process's virtual memory regions, including heaps, stacks, just-in-time (JIT) compiled pages, and dynamically loaded libraries. By remaining entirely in user space, Frida avoids the need for kernel modifications or exploit-based privilege escalation, making it particularly suitable for controlled, research-oriented analysis of Android applications.

Building on this framework, Fridump implements a pragmatic, read-only procedure for per-process memory acquisition. It leverages Frida's APIs to parse /proc/<PID>/maps, identify readable virtual memory mappings, and extract each region in bounded chunks to prevent host memory saturation. Each mapping is then written to a canonical binary artifact, whose filename encodes the package identifier, PID, virtual start/end addresses, access permissions, and corresponding pathname. Fridump can attach to a running process via frida-server on rooted devices or via an embedded frida-gadget within a repackaged APK. In both configurations, its read-only, user-space operation minimizes interference with the system while allowing investigators to recover transient, decrypted, or otherwise volatile data directly from memory. In the non-root configuration, Fridump operates through an embedded frida-gadget. The investigator integrates libfrida-gadget.so (compiled for the target ABI and matched to the client version) within the target application package, ensuring its early loading (e.g. through System.loadLibrary("frida-gadget") in Application.onCreate or an equivalent smali insertion). This configuration allows host-side attachment to the process without device-level root but requires reverse engineering and repackaging of the target APK. For this reason, it is primarily suited to case-specific or small-scale investigations where kernel access is unavailable or legally restricted.

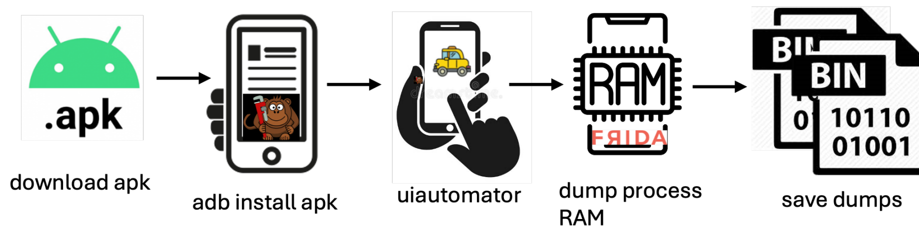

Figure 6.5: The picture shows the acquisition pipeline of a target process, from APK installation, automatic startup, Frida installation, saved dumps from Fridump

The proposed acquisition methodology in Figure 6.5, aims to obtain a time-synchronised, process-specific representation of an application's readable virtual address space. Here I present AndroMemDump - Process RAM to answer to the research question ***How to automatically acquire process RAM with root permissions without kernel recompilations?***

The target application is driven to a predetermined, stable execution state through a deterministic automation framework (e.g. scripted UI interactions that perform installation, login where applicable, and permission grants). Upon reaching this state, the acquisition trigger is executed to guarantee event-aligned and reproducible captures. Immediately beforehand, process metadata (including PID, uptime, and the complete content of /proc/<PID>/maps) is recorded and hashed to permit later correlation between virtual addresses, mapped files, and symbolic information. Fridump then enumerates all virtual mappings, parsing /proc/<PID>/maps to ensure accurate boundaries and offsets, and proceeds to read only regions that expose read permissions. Each readable region

is acquired in deterministic, bounded chunks via `memory.readByteArray` and written to an individual binary file whose filename encodes all relevant metadata (package identifier, PID, start–end addresses, permissions, pathname). The raw mapping binaries constitute the primary evidential artifacts and are preserved unaltered. Secondary artifacts (e.g. string extractions, byte-to-image transformations, or audio reconstructions) are explicitly labelled as derived products and accompanied by metadata documenting transformation parameters and tool versions. `Fridump`'s segmented, per-mapping acquisition significantly reduces the amount of irrelevant data (i.e. kernel structures, unrelated process memory) compared to full physical memory dumps, thereby improving analytical signal-to-noise for application-level artifacts such as heap objects, in-memory plaintexts, and mapped resources. Because captures are performed at runtime, `Fridump` reveals live, decrypted states that static analysis or offline snapshots cannot access. Its scriptability supports fine-grained control, selective region acquisition, on-the-fly hashing, and integration with automation frameworks, enabling deterministic and event-synchronised captures. The purely user-space, read-only approach avoids inserting kernel modules or executing privileged exploits, thus reducing the risk of contaminating evidential state.

However, `Fridump` is inherently limited to the address space of the attached process: it cannot access kernel memory, other processes' heaps, or hardware-backed key material (e.g. secrets stored in the TEE). The act of instrumentation itself may induce observable timing or behavioural deviations (observer effect), and anti-instrumentation mechanisms can hinder or block attachment. In the `frida-gadget` configuration, practical scalability is constrained by the need to repackage target applications, ensure ABI compatibility, and maintain strict version parity between gadget and client components. Moreover, certain applications protected by integrity checks or repacking defenses cannot be instrumented at all. By contrast, full physical memory acquisition tools such as `LiME` (Linux Memory Extractor) provide a complete system-wide snapshot, including kernel space, all user processes, and inter-process structures. This makes `LiME` indispensable when the objective is to reconstruct system state, analyze kernel-level data structures, or correlate evidence across multiple processes. Nonetheless, `LiME` requires elevated privileges and the insertion of a kernel module, operations that are increasingly restricted on modern Android devices and that carry a higher risk of altering system state. It also generates substantially larger data volumes, increasing storage and processing overhead and complicating reproducibility across runs.

In summary, `Fridump` represents a practical, minimally invasive, and reproducible solution for per-process memory acquisition in application-centric investigations, whereas `LiME` remains preferable for comprehensive system-level forensics. When feasible, both methods can be employed in a complementary fashion: `Fridump` for targeted, event-aligned acquisition of volatile application data, and `LiME` for broader contextual reconstruction of the operating environment. The combination offers a balance between precision, evidential completeness, and methodological defensibility.

This work with the image encoding has been sent to an important Journal for peer-review evaluation.

To conclude, by answering to the main research question that lead to this Chapter, the RAM can be acquired with two main frameworks personally developed: `MoLIFE` to guarantee forensics soundness and `AndroMemDump` in the version for complete full RAM and target process RAM.

Approach	**Research Question**	**Contributions**	**Results**	**Limitations**
MoLIFE	How to forensically acquire an Android device without root permissions?	Framework to monitor a critical Android device (preventive, live and post-mortem forensics) to acquire data without root privileges	Acquisition on the mDT is >95% similar to real rooted device; Synchronization to retrieve live data without root	No tests on real forensics investigations
Full RAM	How to acquire the full RAM of an Android device?	Define a procedure to recompile the Android kernel to dump the full RAM and create a profile compatible with Volatility 2 and Volatility 3	Procedure compatible with 80% OS and kernel versions	No tests on real device and recent kernel/OS versions
Process RAM	How to automatically acquire process RAM with root permission without kernel recompilation?	Define a light efficient approach to dump the RAM allocated to a target process (APK)	Extract the memory of 100% malware with anti-analysis techniques	Need of root to dump the memory

Table 6.1: Memory Forensics Contributions

Chapter 7

Detecting Android Malware through Memory Analysis

After memory acquisition and extraction, it is fundamental to analyse the dump and extract meaningful and useful data (i.e. Indicators of Compromise - IoCs), i.e. ***How to analyse the acquired memory?*** The analysis, especially large-scale, can be lead by AI algorithms. In order to have a quick and accurate detection, AI models are needed. For this reason, different encodings have been proposed, but ***What is the most efficient encoding?*** I start analysing images, audio (only a subset of APK due to our computational resources limitations), strings with the same dataset of APK with target-process memory dumps. Subsequently, I adopted the widely known and used `Volatility` tool for complete RAM analysis on a subset of APK (i.e. dropper and fakeapps). I will pass each encoding to specific ad-hoc algorithms and interpret their results. I analyse the dumps using different tools and encodings i.e. `Volatility` output, strings, images and audio. This Chapter demonstrates how Android Memory Forensics can be a valid alternative for malware detection and analysis, with high accuracy on stealthy attacks such as malware with anti-analysis techniques, obfuscation, adversarial samples and stegomalware, i.e. ***Can memory forensics analysis be used to detect evasive malware?***

7.1 Memory Analysis with Image Encoding

Binary-to-image encoding is widely used in the literature, especially for malware analysis, but *Can we convert the dumped memory region in visual images?* Early studies such as Nataraj *et al.* [145] demonstrated that mapping raw binary bytes to pixel intensities yields characteristic visual patterns that reflect code structure, entropy, and obfuscation, forming the foundation of image-based malware classification. Subsequent frameworks (e.g. `DexRay [68], DiDroid [160], cRGBMem [29]`) extended this principle to Android environments by converting `classes.dex` or other APK binaries into 2D grayscale or RGB matrices, later processed by CNN or Vision Transformers for automated feature extraction. While these approaches primarily focus on static artifacts (e.g. bytecode, manifest files, or embedded resources) recent research has shifted toward representing volatile memory data as images, bridging DF and DL. Works such as `crGBMem` [29] highlight that encoding process or system memory into image form exposes transient runtime evidence, such as decrypted payloads or dynamically loaded modules, which cannot be captured through static inspection. In this context,

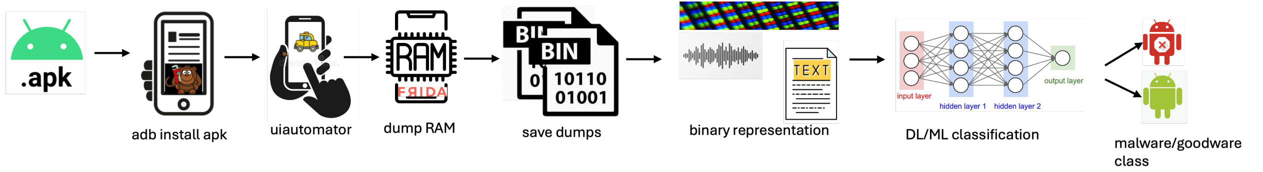

Figure 7.1: Fridump analysis pipeline using image, audio or text encoding and xAI to retrieve the most significant patterns in the encoding, mapping back to the memory bytes

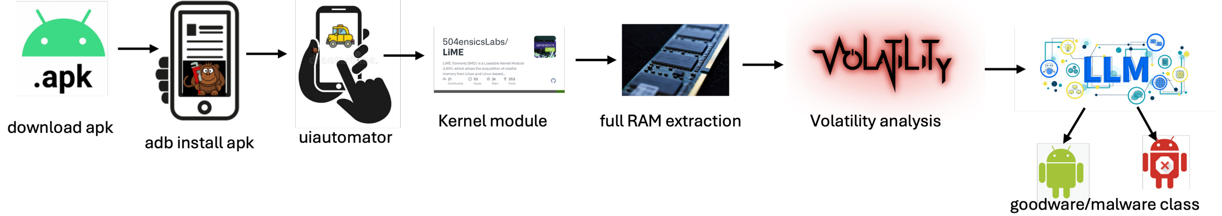

Figure 7.2: Full RAM analysis pipeline using Volatility tool and LLMs for data explanation

Figure 7.3: Comparative visualization of target process and complete RAM analysis

memory-based binary-to-image conversion provides a fine-grained and temporally aware representation of Android malware behavior, allowing models to detect hidden or obfuscated activity that emerges only at runtime. The combination of visual encoding and memory forensics thus represents a promising direction for robust, adversarial-resilient Android malware detection.

Still, all the works face a problem about 2D image encoding from binary data, i.e. spurious correlation between near vertical pixels which do not represent near memory bytes as highlighted in `DexRay` [68]. For this reason, both 2D encoding and 1D image encoding (a new developed algorithm) were applied. In fact, one of the main contributions is the comparison between classification results using both 1D and 2D images.

The adopted binary-to-image approach [235] converts the byte stream of a memory dump

$$\mathcal{B} = \{b_1, b_2, \dots, b_n\}, \qquad b_i \in [0, 255] \tag{7.1}$$

into visual representations in either grayscale or RGB format.

In the 2D layout, bytes are grouped (*one per pixel* in grayscale, or *triplets per pixel* in RGB) and rasterized in *row-major order* to fill an image of width W and height $H = \lceil \frac{N}{W} \rceil$. The 2D image is defined as

$$I_{2D}(r, c) = p_{(rW+c)+1}, \qquad 0 \le r < H,\ 0 \le c < W \tag{7.2}$$

with height and coordinates $H = \lceil \frac{N}{W} \rceil, \quad r = \lfloor \frac{i-1}{W} \rfloor, \quad c = (i-1) \bmod W.$

The pixel sequence fills the image left to right along each row, continuing to the next row after every W pixels. The width W is selected according to the heuristic proposed by Nataraj [145] (*e.g.*, $W \in \{32, 64, 128, 256\}$), ensuring visually interpretable aspect ratios and comparability across samples of different sizes. This mapping produces structured textures where code sections, headers, and compressed blocks appear as distinct visual patterns. Altought the 2D wrapping introduces artificial spatial adjacency, bytes that are sequential in memory may become vertically separated, while vertically adjacent pixels may correspond to distant addresses. As a result, 2D images enhance interpretability and compactness but distort the true linear continuity of the original binary.

To eliminate such artifacts, 1D layout has been created, where all bytes are placed sequentially along a single horizontal axis:

$$I_{1D}(0, c) = p_{c+1}, \qquad H = 1, \quad W = N \tag{7.3}$$

Each pixel directly encodes the next byte (or RGB triplet) in order, producing a strictly isomorphic mapping between the byte sequence and pixel sequence. Unlike the 2D case, the 1D representation preserves true byte-to-pixel continuity, ensuring that neighboring pixels correspond exactly to neighboring memory addresses. Although the 1D format loses the compact and texture-rich layout of 2D images, it provides a semantically accurate, lossless visualization of sequential memory behavior, more suitable for analyzing ordered structures such as instruction streams or entropy gradients.

In summary, 2D images prioritize visual interpretability and exploit mature 2D CNN architectures, while 1D images maintain structural fidelity to the raw memory layout, avoiding spurious correlations introduced by 2D wrapping. The two encodings thus complement each other: 2D layouts favor high-level pattern discovery, whereas 1D layouts preserve the precise sequential relationships inherent in volatile memory data.

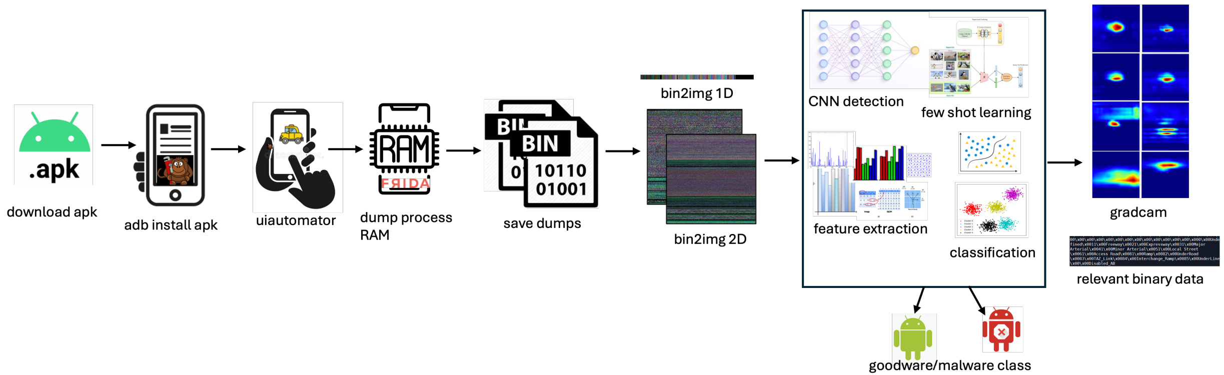

Figure 7.4: The picture shows the pipeline for analysing the dump of a target-process with image encoding, using 1D or 2D images, passed in input to proper CNN or extracting desired features and classifying with ML algorithms. In the last stage, after applying Grad-CAM, the most significant pixels are extracted and mapped back to the memory bytes to retrieve the most important memory patterns

All converted RGB images are analyzed using both ML and CNN approaches. Classification operates at the APK level: region-level predictions are first obtained and then aggregated per application (mean aggregation), with a decision threshold determined from the ROC curve using Youden's J statistic [76, 225].

For traditional ML, a comprehensive feature extraction pipeline is applied to 2D images following prior literature [145]. Extracted descriptors include color histograms and moments, GLCM texture metrics, Local Binary Patterns (LBP), Histogram of Oriented Gradients (HOG), edge and shape descriptors (Sobel, Hu moments), global grayscale statistics, and FFT-based frequency features. These complementary features capture color, texture, spatial, and frequency properties of memory-region images, providing a rich input space for algorithms such as SVM, KNN, Random Forest, and Decision Tree classifiers.

In parallel, CNN-based approaches are used for automatic feature learning. For 2D images, transfer learning is applied to pre-trained architectures (ResNet, EfficientNet, MobileNet, ConvNeXt,

ViT, Inception, and the implementations found in DiDroid [160], DexRay [68], cRGBMem [29]), replacing the classification head with a binary layer (benign vs. malicious). The training pipeline handles region parsing, normalization, and APK-level evaluation using metrics such as ROC-AUC, PR-AUC, and confusion matrices.

For 1D images, the ResNet-18 architecture has been adapted into a one-dimensional variant, preserving the convolutional backbone up to the final convolutional block. The model processes each memory-region image as a 1D signal, where the horizontal axis encodes the true byte order. A modified pooling stage collapses the vertical dimension via adaptive average pooling configured as (1, None), preserving the horizontal continuity of the feature maps. This produces a sequence of embeddings with shape [batch, 512, T], where T denotes the horizontal extent. A lightweight classification head then refines these embeddings using a 1D convolution to capture local byte dependencies, followed by adaptive max pooling and a fully connected multi-layer perceptron (MLP) with dropout regularization. This architecture retains the representational power of pretrained 2D backbones while aligning the network with the sequential structure of memory dumps.

In addition to the ResNet-based variant, a fully custom one-dimensional CNN explicitly built for horizontally serialized RGB memory dumps has been designed. The model accepts input sequences of length 1024 with three channels (RGB) and applies a hierarchy of convolutional blocks (`Conv1d → BatchNorm1d → ReLU → Pooling`) that progressively expand feature dimensionality ($64 \rightarrow 128 \rightarrow 256$) while reducing sequence length. adaptive max pooling layer normalizes the output to a fixed length of 32, ensuring consistent input dimensions for the classifier regardless of the original memory size. A simple attention mechanism, implemented via a 1×1 convolution followed by a sigmoid activation, reweights feature channels to emphasize the most discriminative temporal patterns before classification. Finally, the classifier projects the flattened sequence into a 128-dimensional latent space with dropout regularization, followed by the final binary output layer. This architecture treats each memory dump as a structured 1D signal, enabling efficient learning of byte-wise dependencies without relying on 2D spatial priors or pretrained natural-image features.

The experimental evaluation of this approach demonstrates that volatile memory analysis can be an effective and interpretable means for Android malware detection. Using a dataset of $2,073$ malware samples from multiple families and $2,279$ benign APK both from different Android API levels, the memory structure has been profiled first, observing that the majority of the allocated RAM is occupied by system libraries and compiled runtime artifacts rather than raw APK files. From these dumps, over five million RGB and grayscale images has been produced, each representing individual memory regions. Across experiments, the most informative memory areas were consistently the data section (i.e. the data space allocated to the target APK) combined with the stack region, as these contain both persistent code artifacts and dynamic runtime objects, as shown in Figure 7.5. When comparing pre-trained and custom CNNs (Figure 7.8), ResNet-18 achieved the best trade-off between accuracy, interpretability, and computational cost, surpassing both deeper architectures and state-of-the-art baselines such as DexRay, DiDroid, and cRGBMem. Feature-based models using handcrafted descriptors (e.g. color histograms, GLCM, LBP, HOG, FFT) were less effective, as seen in Figure 7.7, while few-shot learning on CNN-derived embeddings improved robustness as shown in the comparison heatmap in Figure 7.6.

The study also examined the impact of image representation (detailed in Table 7.1): RGB encoding slightly outperformed grayscale, and the novel 1D RGB layout achieved the highest true

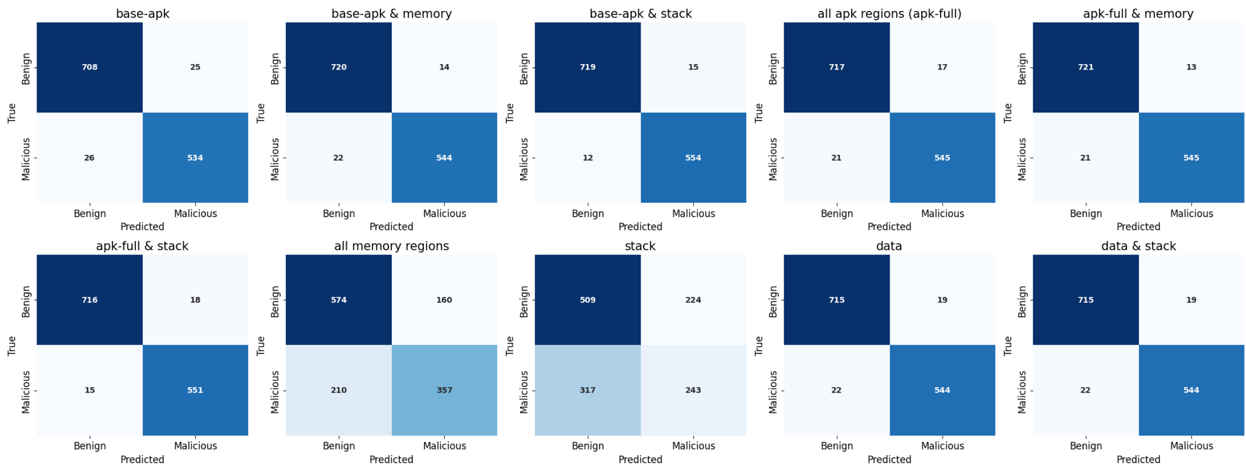

Figure 7.5: Confusion Matrix of Different Memory Areas using 2D images with pre-trained ResNet18

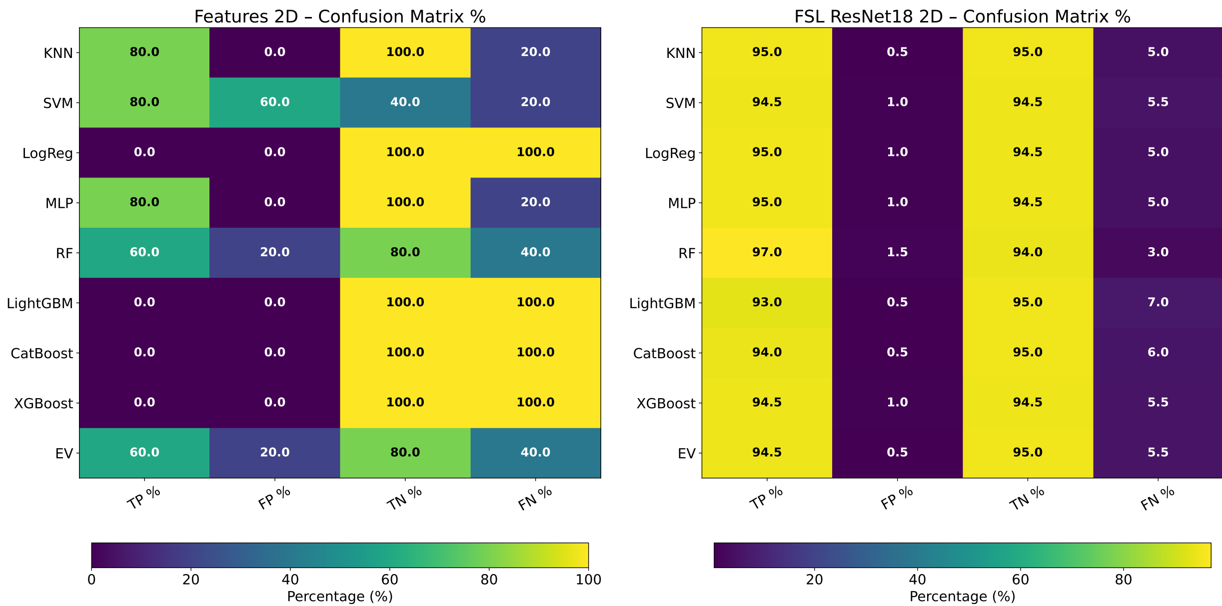

Figure 7.6: Comparison of Machine Learning algorithms using feature-based classification and Few Shot Learning in the last ResNet18 CNN

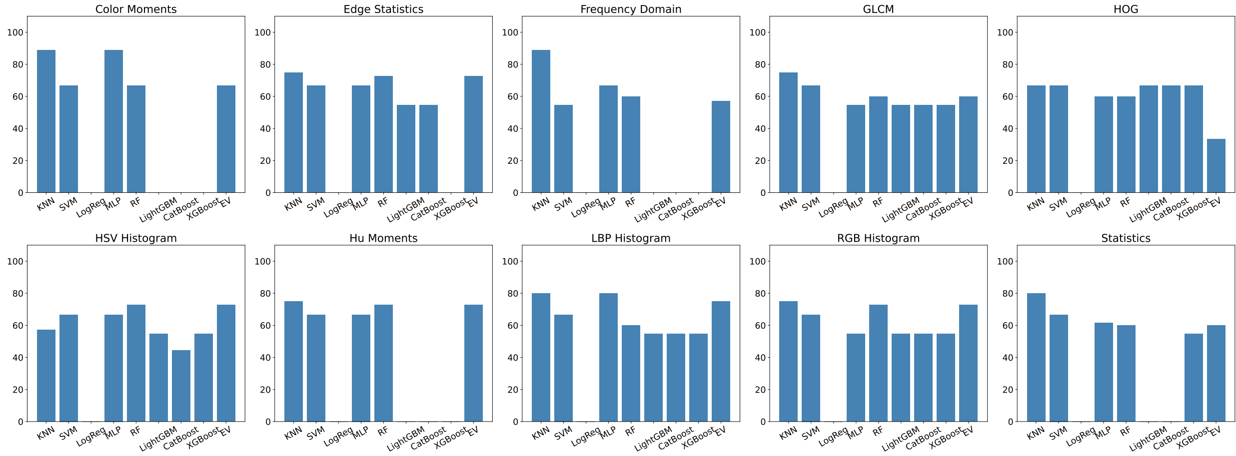

Figure 7.7: Details of Feature classification for 2D RGB images. F1 score for each group of features (from top left: color moments, edge, frequency, GLCM, hog, HSV, Hu, LBP, RGB, Statistics) classified with fine tuned algorithms (from left to right: KNN, SVM, LogisticRegression - LogReg, MLP, Random Forest - RF, LightGBM, CatBoost, XGBoost, Ensamble Voting - EV).

Mode	Custom1D		ResNet18_1D		ResNet18_2D	
	Benign	Malicious	Benign	Malicious	Benign	Malicious
Baseline	97.81	97.88	98.19	97.52	97.41	96.11
Gray	97.16	98.41	98.45	97.35	96.87	96.64
Remove regions	100.00	16.40	100.00	2.12	99.73	4.60
Remove pixels	99.61	20.00	100.00	0.00	99.46	42.30
Classify regions	97.94	81.06	99.87	0.00	99.86	95.22
Classify pixels	98.19	41.24	100.00	4.25	100.00	1.42

Table 7.1: TPR (%) for Benign and Malicious across CAM modes (removing or classifying only most significant memory regions and pixels) and models (Custom1D, Resnet1D, Resnet18) according to the different encodings (1D and 2D, both RGB and grayscale)

positive rate (about 98%) by avoiding the spurious vertical correlations found in 2D images. Grad-CAM visualizations confirmed that stack, cache, code, and native library sections were the most influential for classification, and by mapping highlighted pixels back to raw bytes, analysts could recover clear-text variables, API calls, and cryptographic keys. About native code, the relation with vulnerabilities or the use of specific products has been checked. We discovered that some libraries are loaded dynamically because the `lib.so` is found in the RAM but not in the lib directory of the APK; sometimes because of anti-analysis techniques, the library is not found in the lib directory but in the assets. For the available libraries, some the classification in some trojan samples depends on a high risk of vulnerabilities in the native code by adopting the approach described in 5.1. More detailed studies must be developed as those described in 5.2 and 5.3 with also a more reliable library association technique for native code identification, hence vulnerability attribution.

A temporal drift experiment, training on 2017–2018 samples and testing on 2022 malware, showed strong cross-version generalization (about 97% accuracy), while time-based acquisitions over app runtime confirmed that continuous RAM monitoring improves detection of late-loading payloads. In comparative testing, the presented RAM approach successfully detected over 350 adversarial and anti-analysis malware samples that current state-of-the-art tools (i.e.`Drebin`, `Didroid`, `Dexray` and `Entroplyzer`) failed to classify. This demonstrates memory forensics resiliency to obfuscation, reflection, and runtime module loading, shown in Figure 7.9. Detailed case studies on target APK (e.g. Chrome, WhatsApp, Avast, KleePrank malware) have been compared to

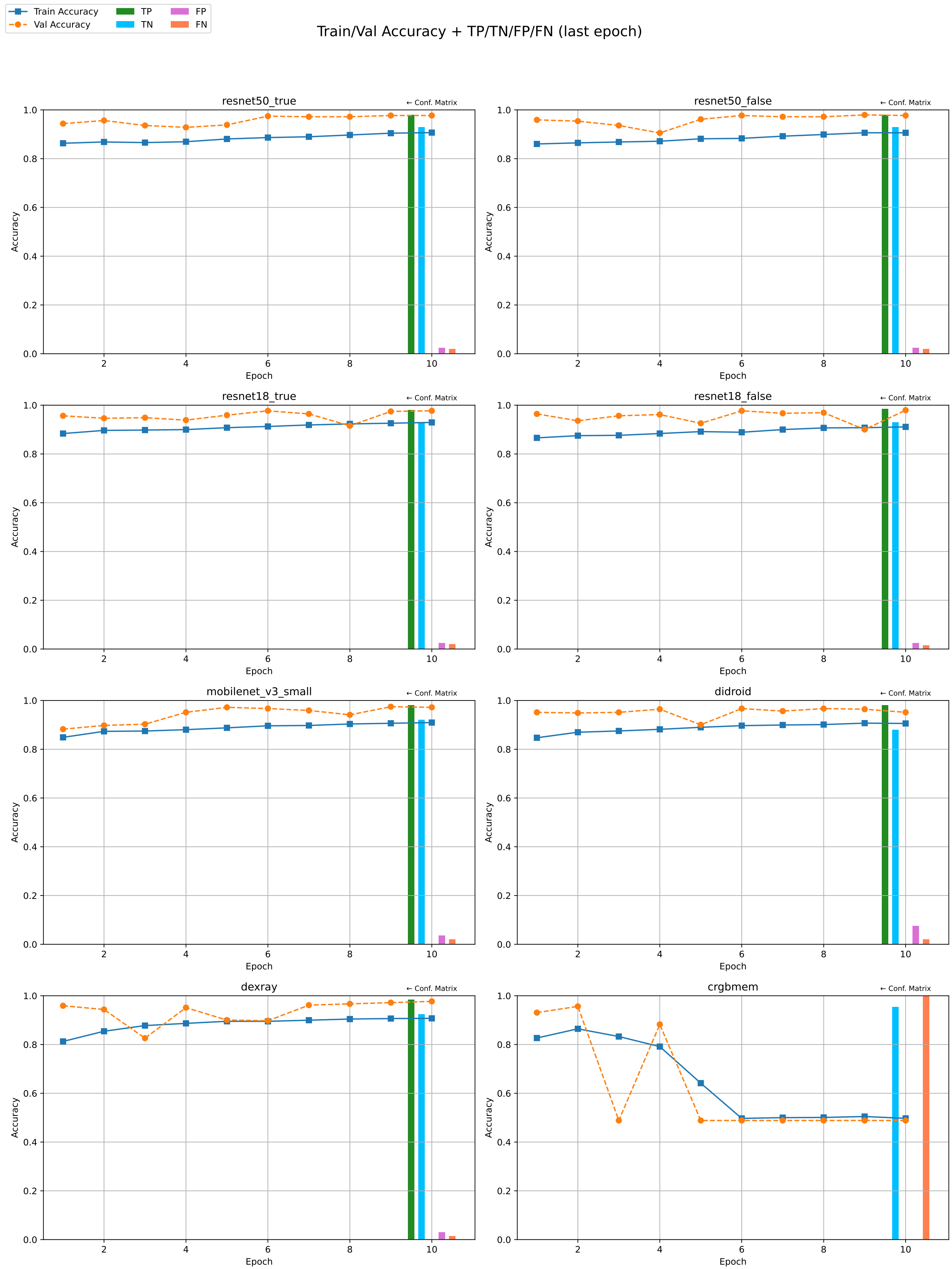

Figure 7.8: Training performances over 10 epochs for popular state-of-the-art CNNs and confusion matrix of the last validation epoch

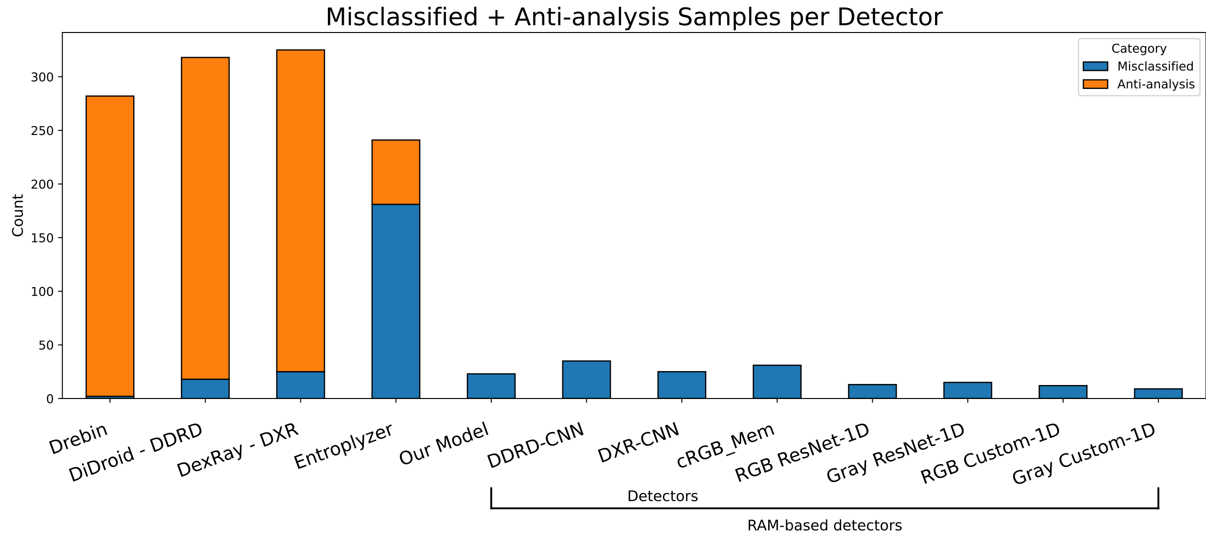

Figure 7.9: The picture shows the number of misclassified for each detector, including RAM approach using the different CNNs and image encodings

`VirusTotal` and `MobSF`, showing that RAM dumps reveal live clipboard data, decrypted payloads, network endpoints, and hidden modules invisible to static or dynamic tools. Together, the three approaches offer a more complete picture of malware behavior. Overall, results confirm that the presented RAM approach bridges the gap between static and dynamic analysis, offering a complementary forensic layer that exposes real-time, memory-resident evidence of malicious behavior, with strong generalization across time, architectures, and obfuscation strategies.

This work demonstrates that memory regions can be visualized into traditional colored images and passed to CNNs for an accurate detection.

7.2 Memory Analysis with Audio Encoding

Recent research on malware detection has explored unconventional data representations that leverage the perceptual and analytical power of the audio domain, but ***Can we convert the dumped memory region in audible sound?*** Malware sonification refers to the process of transforming binary or behavioral data extracted from malicious software into audio signals, enabling the application of digital signal processing and ML techniques originally designed for speech and music analysis. This emerging paradigm seeks to exploit the structural regularities and entropy patterns inherent in executable code, which, when mapped into acoustic space, yield distinctive spectral and temporal signatures. Early studies introduced binary-to-sound conversion as a novel feature extraction method for malware classification, detailed as follows. In one of the first contributions, Farrokhmanesh and Hamzeh [74] proposed converting Windows executables into MIDI representations, demonstrating that musical mappings of opcode sequences could reveal family-specific rhythmic and harmonic patterns. Subsequent work extended this principle to Android malware: Mercaldo and Santone [136] converted `.dex` files into WAV audio, applying Mel-Frequency Cepstral Coefficients (MFCCs), chroma, and other spectral descriptors to train classical machine learning models. Their findings showed that acoustic features effectively separate malware from benign applications, with

accuracies comparable to traditional static analysis. Building on these foundations, later studies focused on optimizing feature extraction and reducing computational complexity. Kural *et al.* [118] proposed `APK2Audio4AndMal`, a framework that extracts over thirty audio features and applies feature selection algorithms to identify the most discriminative descriptors for malware family classification. Similarly, Tarwireyi *et al.* [200] introduced a multi-feature fusion approach, combining tonal, spectral, and temporal features before training gradient-boosting models such as XGBoost, achieving near-perfect accuracy on benchmark datasets. Despite promising results, these approaches remain predominantly static, operating directly on the bytecode or APK content without considering runtime behavior. As a consequence, such static approaches fail to capture dynamic phenomena such as code unpacking, self-modification, or encrypted payload activation, which are central to modern obfuscated malware. To address these limitations, emerging research proposes extending sonification to runtime memory representations, transforming live memory regions into audio signals to encode both structural and behavioral characteristics. This transition from static binaries to dynamic memory sonification introduces a richer and temporally aware representation of application behavior, enabling the detection of malicious patterns that manifest only during execution.

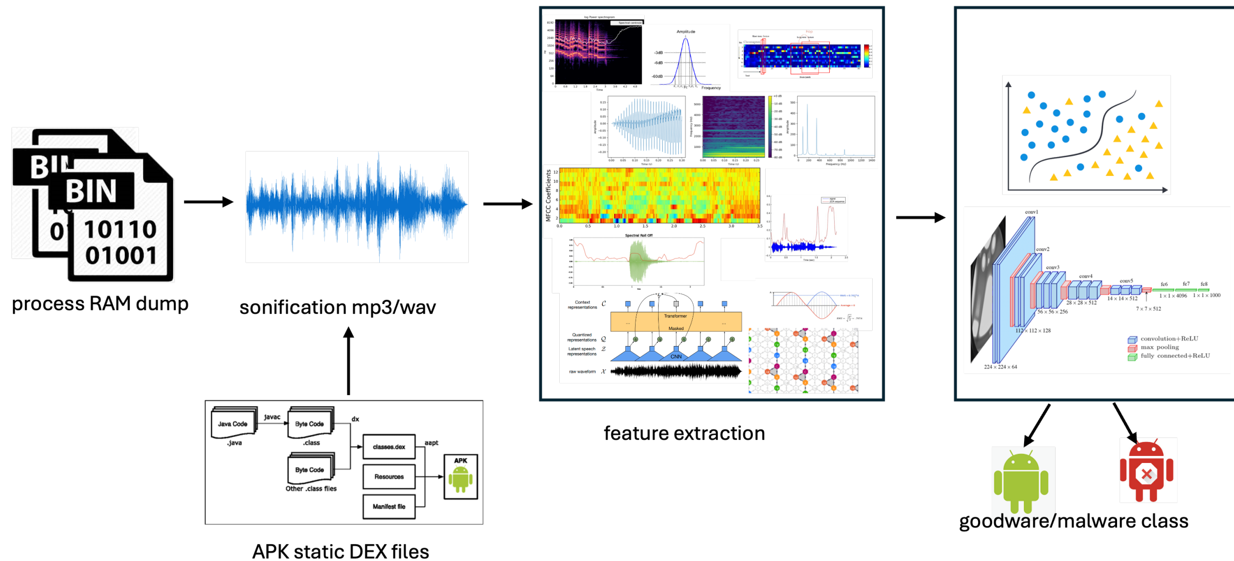

Figure 7.10: The picture shows the pipeline for analysing the dump of a target-process with audio encoding, comparing the dynamic analysis with the static analysis. The memory dumps and the DEX files for each APK are sonified with MP3, WAV8 and WAV16, proper signal features are extracted and a classification with ML and CNN is made

The proposed approach extends traditional audio-based malware classification to the dynamic domain by introducing a memory forensics analysis on the dump of the allocated target process. The sonification process represents the central methodological innovation of this study, translating binary code and memory artifacts from Android applications into structured audio waveforms suitable for digital signal analysis and ML classification. This transformation is grounded in the principle of direct sonification, which establishes a bijective mapping between the raw byte stream of an executable or memory snapshot and the amplitude values of a one-dimensional digital audio signal, similarly to what done for 1D images. Unlike semantically driven representations (e.g. opcode graphs or disassembly trees) this approach preserves the low-level statistical texture of binary data, thereby enabling the detection of latent structural regularities that are otherwise invisible to traditional

static or dynamic code analysis.

Binary sonification draws inspiration from the field of data-driven auditory display, wherein non-audio data are encoded as sound to reveal patterns through acoustic or spectral interpretation. In this context, malware binaries and runtime memory segments are treated as digital signals, whose byte-level fluctuations encapsulate underlying program logic, data entropy, and obfuscation patterns. By mapping these fluctuations directly onto an audio amplitude scale, the process converts machine-level structures into spectral representations that can be analyzed through well-established audio processing techniques such as Fourier analysis, spectrogram decomposition, and cepstral coefficient extraction. This mapping is not arbitrary: empirical evidence shows that malware samples exhibit repetitive and high-entropy sequences, which correspond to characteristic harmonic bands and rhythmic textures in the frequency domain, providing a discriminative signature between benign and malicious behaviors.

The proposed pipeline integrates both static and dynamic acquisition modes to evaluate the discriminative potential of different data sources. In the static phase, APK are decompressed to extract the `classes.dex` file. This component is selected as it provides a compact and self-contained representation of the application's operational behavior at rest, independent of runtime factors such as dynamic code loading or heap allocation. The extracted bytecode is then passed directly to the sonification module as a continuous binary stream, ensuring that the mapping preserves the original byte order and internal code structure. Static analysis thus serves as a baseline for evaluating how much discriminative information can be derived from static executables alone, in contrast to runtime memory regions used in the dynamic approach.

In the dynamic analysis, memory dumps are obtained by reading the process mappings from `/proc/<pid>/maps` (with the `Fridump` acquisition technique 6.2.3) and selecting relevant regions (e.g., `[stack]`, `base.apk`, shared libraries) through regular expressions. Each selected region is read as a binary sequence and concatenated to form a contiguous byte stream representing the target memory view. This approach enables the capture of runtime structures such as dynamically loaded classes, decrypted payloads, and transient execution artifacts that are not visible through static inspection.

Given a byte stream $\mathcal{B} = \{b_1, \ldots, b_n\}$ with $b_i \in [0, 255]$, the corresponding audio waveform is defined as

$$A(t_i) = \frac{b_i - 128}{128}, \qquad 0 \leq i < N \tag{7.4}$$

for 8-bit PCM encoding, and as

$$A(t_k) = \frac{\text{int16}(b_{2k-1} + 256\, b_{2k})}{32768}, \qquad 0 \leq k < \left\lfloor \frac{N}{2} \right\rfloor \tag{7.5}$$

for 16-bit PCM (little-endian) encoding, where each $A(t)$ denotes the normalized amplitude sample corresponding to the original byte sequence. This transformation ensures that the full range of byte intensities is represented in the audio waveform while maintaining a one-to-one correspondence with the original binary stream. For the 16-bit encoding, adjacent byte pairs are concatenated into signed 16-bit integers (in little-endian order), thereby increasing the quantization depth and effective signal resolution.

To systematically evaluate the influence of encoding fidelity and compression on classification accuracy, three audio formats were implemented. The first configuration, WAV 8-bit (32 kHz, mono),

provides a direct byte-to-sample mapping with minimal transformation, preserving byte-level detail and offering simplicity and reproducibility. This format was found to generate highly discriminative spectral features while maintaining low computational cost. The second configuration, WAV 16-bit (32 kHz, mono), extends the bit depth to capture finer amplitude variations and increase dynamic range. Although theoretically capable of representing subtler signal gradations, empirical results indicated that the additional resolution introduced redundant noise, slightly degrading classification performance. Finally, the MP3 (192 kbps, 19–32 kHz, mono) encoding applies perceptual compression to evaluate the robustness of the classification pipeline against lossy encoding. Despite discarding certain high-frequency components, MP3 preserved the discriminative spectral structures characteristic of malware samples while reducing file size by approximately 25% and overall processing time by nearly 50%. The output of this phase consists of a set of labeled audio files organized by sample class (e.g. benign, adware, banking malware). Each file encapsulates a reproducible acoustic projection of the corresponding binary or memory region, preserving the original byte order and internal structure.

Preliminary spectral analyses using Short-Time Fourier Transform (STFT) revealed consistent differences between benign and malicious samples. Malicious applications displayed dense and periodic spectral peaks concentrated between 2–14 kHz, corresponding to repeated byte patterns typical of obfuscation, encryption routines, or packed payloads. In contrast, benign applications exhibited smoother broadband spectra with lower spectral variance, reflecting the more regular structure of unencrypted code and resource data. These observations confirm that direct sonification preserves meaningful structural information and that malware-related entropy manifests as distinct acoustic signatures in both temporal and frequency domains.

The experimental evaluation demonstrated the effectiveness of the proposed dynamic memory sonification methodology for Android malware detection, revealing several key findings across both static and dynamic analysis stages. In the static experiments in Table 7.2 performed on the `CIC-Maldroid2020` dataset [132], the system reproduced state-of-the-art results with minor variations, achieving a peak accuracy of 93.9% using 8-bit WAV encoding and manual feature extraction via `librosa` combined with a Random Forest classifier. The 16-bit WAV and MP3 encodings produced slightly lower accuracies (92.3% and 92.6%, respectively), confirming that increased bit depth or lossy compression does not necessarily improve discriminative power. Although the `Wav2Vec2` transformer reduced feature extraction time by over 60%, its accuracy (91.5%) was marginally below that of the manual approach, demonstrating a trade-off between computational efficiency and performance.

Model	Accuracy (%)	Benign F1 (%)	Mal. F1 (%)	FP	FN
Random Forest	97.0	97.0	97.0	1	3
SVM	96.0	96.0	96.0	2	3
Voting Ensemble	95.0	95.0	95.0	0	6
Logistic Regression	94.0	94.0	94.0	4	3
MLP	94.0	94.0	94.0	3	4
KNN	93.0	94.0	93.0	0	8

Table 7.2: Classification performance on static `APK` sonification from dynamic dataset

The dynamic memory sonification experiments achieved notably higher detection performance.

Using memory dumps acquired during application startup, the system reached a maximum accuracy of 98.0%, with zero false positives and a single false negative across seven distinct memory region selection strategies. The best-performing configuration combined execution context and code regions (the apk_full_stack strategy), indicating that the joint representation of the application's stack, loaded libraries, and Dalvik structures captures runtime behavior that static analysis cannot observe. Isolated regions described in Table 7.3, such as the stack alone (90.0%) or complete memory regions (93.0%), produced inferior results, highlighting the importance of selective region inclusion.

Strategy	Encoding	Model	Accuracy (%)	FP	FN
all_data_paths	MP3	Random Forest	98.0	0	1
apk_full	MP3	Random Forest	98.0	0	1
apk_full_memory	MP3	Random Forest	98.0	0	1
apk_full_stack	MP3	Random Forest	98.0	0	1
base_apk_stack	MP3	Random Forest	98.0	0	1
base_apk_memory	MP3	Random Forest	98.0	0	1
data_stack	MP3	Random Forest	98.0	0	1
base_apk	WAV16	XGBoost	97.0	0	2
complete_memory_regions	WAV16	CatBoost	93.0	1	3
stack	WAV8	Random Forest	90.0	5	1

Table 7.3: Best classification results for each memory region selection strategy.

Across all experiments, WAV 8-bit encoding provided the most balanced results as shown in Table 7.4, with the highest mean accuracy (95.8%) and minimal false prediction rates, while MP3 offered comparable performance with substantially reduced storage and computation overhead. Classifier comparison further confirmed that ensemble methods (i.e. Random Forest, XGBoost, CatBoost, and LightGBM) consistently outperformed single models, achieving stable results near 98% accuracy.

Encoding	Avg Accuracy (%)	Avg F1-Score (%)	Avg FN	Avg FP
WAV8	95.8	95.8	1.64	0.81
WAV16	93.3	93.3	2.56	1.39
MP3	92.6	92.4	3.24	1.07

Table 7.4: Average classification performance for each audio encoding format across all configurations.

In conclusion, the results validate the viability of audio-based malware classification through dynamic memory sonification. The approach effectively bridges binary-level structures and spectral audio analysis, capturing runtime artifacts that static representations miss. The findings demonstrate that lightweight, information-preserving sonification of memory data enables near-perfect malware discrimination while reducing computational complexity. These results position dynamic sonification as a promising complementary tool for modern Android malware forensics and real-time behavioral analysis, with future work directed toward larger datasets, multi-phase memory acquisition, and integration of transformer-based embeddings for enhanced scalability and generalization.

Model	Accuracy (%)	Benign F1 (%)	Mal. F1 (%)	Avg FN	Avg FP
CNN1D	98.0	98.0	98.0	1.00	0.00
CatBoost	98.0	98.0	98.0	1.00	0.00
LightGBM	98.0	98.0	98.0	1.00	0.00
MLP	98.0	98.0	98.0	1.00	0.00
Random Forest	98.0	98.0	98.0	1.00	0.00
SVM	98.0	98.0	98.0	1.00	0.00
Voting Ensemble	98.0	98.0	98.0	1.00	0.00
XGBoost	98.0	98.0	98.0	1.00	0.00
Logistic Regression	97.7	97.7	97.7	1.33	0.00
KNN	95.3	95.7	95.3	2.67	0.00
Mean	**97.7**	**97.7**	**97.7**	**1.20**	**0.00**

Table 7.5: Classification performance for `apk_full_stack` memory region across all classification models. Performance metrics represent averages across three audio encoding formats.

With respect to image-based encoding, audio analysis shows a comparable yet more lightweight and computationally efficient alternative for volatile-memory malware detection. While image representations seen previously, achieve strong interpretability and up to 98% accuracy through spatial CNN processing of millions of memory-region images, the dynamic memory sonification approach attains a similar 98% accuracy using far smaller audio datasets and simpler ensemble classifiers. Unlike image-based methods that rely on deep convolutional architectures and pixel-level spatial dependencies, sonification transforms memory bytes into compact spectral features that preserve temporal and structural information without heavy preprocessing. Consequently, the audio pipeline reduces feature-extraction cost and storage requirements while maintaining near-identical detection performance. However, image-based analysis provides superior explainability via visual heatmaps and byte-level traceability, whereas sonification offers a more efficient and scalable modality better suited for real-time or resource-constrained forensic environments. In the image classification, the highest accuracy was obtained from the data + stack combination, where static code artifacts and dynamic runtime objects jointly capture execution semantics. Conversely, the dynamic memory sonification experiments identified the apk_full_stack configuration (combining execution context, code segments, and loaded Dalvik structures) as the most discriminative region set, outperforming isolated memory areas such as stack-only or full dumps. Both findings converge on the principle that hybrid memory selections, which integrate static and dynamic components, yield the most informative behavioral signatures. However, sonification achieves this with smaller, information-preserving audio encodings that reduce computational overhead while maintaining the same 98% accuracy, whereas image-based models depend on larger datasets and deeper convolutional networks for comparable results. This work has been sent to a conference and demonstrates that audible sound can be a good encoding for memory regions to detect malicious code at runtime.

7.3 Memory Analysis with Text Encoding

String analysis in Android memory forensics serves as a critical complement to byte-level image representation, providing a semantic lens through which the latent behaviors of malicious applications can be interpreted. Hence, ***Can we read the content of the dumped memory?*** The current state-of-the-art in Android string-based analysis primarily relies on static extraction from

files, APK resources, or disassembled Java bytecode. Tools such as `Androguard` [18], `Apktool`, and `MobSF` parse these files to retrieve embedded Unicode or UTF-8 strings, which are then subjected to rule-based or statistical analysis to identify suspicious constants, URLs, API calls, or cryptographic artifacts. While effective for unobfuscated applications, these approaches are inherently limited by code obfuscation, string encryption, and runtime decryption techniques widely used by modern Android malware families. Obfuscators like `DexGuard` [99] and `ProGuard` [100] dynamically decrypt strings only during execution, rendering static methods incapable of reconstructing them. Similarly, dynamic analysis frameworks based on instrumentation (e.g. `TaintDroid` [73], `DroidScope` [220]) struggle to capture short-lived data that resides in memory only during specific execution windows.

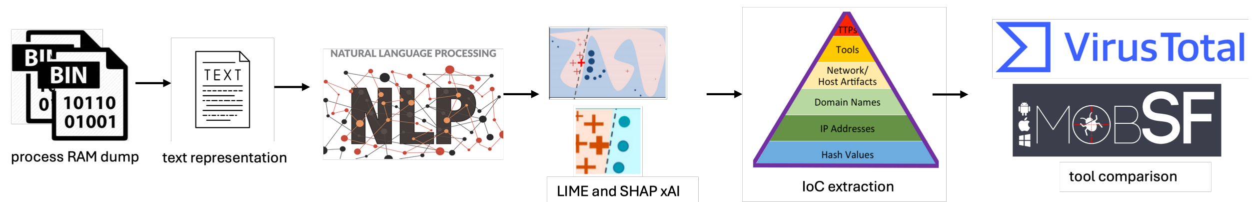

Figure 7.11: The picture shows the pipeline for analysing the dump of a target-process with text encoding, hence extracting the UTF-8 characters from the dump. Subsequently, they are passed to NLP algorithms and xAI algorithms are applied to extract the most significant human-readable text. They represent IoC and are compared with traditional tools VirusTotal and MobSF

To address these challenges, this methodology adopts a memory-resident string extraction pipeline that directly operates on the target-process memory forensics. The process begins by scanning each binary memory region for sequences of printable ASCII or UTF-8 characters that exceed a minimum entropy threshold, thereby filtering out non-textual noise. Each identified string segment is stored alongside its corresponding virtual memory address, allowing traceability between textual features and memory locations. The decoding process supports both UTF-8 and UTF-16 encodings and includes a normalization phase that removes null bytes, control characters, and redundant padding to ensure the correct reconstruction of runtime string buffers.

Extracted strings are then transformed into numerical features through a TF–IDF vectorization scheme. This encoding captures the relative importance of each token across the dataset, emphasizing words that are characteristic of specific malware families while down-weighting common Android or system-related terms. The TF–IDF matrix is subsequently used to train lightweight yet interpretable machine learning models, primarily logistic regression and linear SVMs, for binary malware classification. To enhance transparency, `LIME` (Local Interpretable Model-Agnostic Explanations) and `SHAP` (SHapley Additive exPlanations) have been applied to the trained models, enabling fine-grained interpretation of which tokens most strongly influence the classification decision. Through this explainable analysis, tokens associated with reflection mechanisms (e.g. `java.lang.reflect`, `invoke`, `method`), cryptographic operations (e.g. `AES`, `Cipher`, `Base64`), and communication channels (e.g. `HttpURLConnection`, `Socket`, `URL`) consistently appeared as the most influential features contributing to the “malicious” class prediction. Conversely, benign applications were typically characterized by UI-related or framework-related strings (e.g. `MainActivity`, `layout`, `com.google.android.gms`).

Empirically, the string-based classification achieved an average **accuracy of** 91.2%, with an F1-score of 87.4%, demonstrating that memory-resident strings alone provide a strong discriminative signal even in the absence of byte-level or image-based representations. False negatives primarily

originated from highly obfuscated malware that deferred decryption of strings until late runtime phases or after network communication. Nevertheless, periodic memory snapshots mitigated this issue: when string extraction was performed at three time intervals during execution (i.e. initialization, post-permission grant, and post-network activity), detection improved by nearly 6%. The approach also exhibited strong temporal resilience, maintaining stable performance when evaluated on newer malware samples collected up to five years after the training data, underscoring that runtime strings—unlike static bytecode—encode behavioral semantics that are less affected by API evolution or platform changes.

In terms of explainability and forensic interpretability, mapping the TF–IDF and SHAP outputs back to the memory address space enables analysts to directly locate the regions in which influential strings were stored. Cross-referencing these offsets with Grad-CAM results from the image-based models revealed a strong spatial correlation: the most important string features typically resided in the same memory regions (e.g.`data`, `baseapk` and `stack`) that CNN-based models had identified as salient for classification. This cross-modal consistency reinforces the interpretive reliability of the framework. For example, in one case study involving the *FakeApp* trojan, the most influential string tokens (e.g. `getDeviceId`, `HttpURLConnection`, and `postRequest`) were located in the same stack region highlighted by the Grad-CAM heatmap as contributing to the malicious classification, confirming the presence of reflective network exfiltration routines in the live process memory.

Comparative analysis against state-of-the-art static and dynamic tools further demonstrates the advantages of this RAM-based string analysis. Tools such as `VirusTotal` and `MobSF` successfully extracted only a fraction of the runtime strings, limited to those statically embedded within the file. Even advanced static detectors like `Drebin`, `MaMaDroid`, and `DexRay` failed to detect malware samples employing string encryption or reflection-based payload activation. In contrast, the proposed RAM-level extraction recovered 100% of the decrypted runtime strings from over 350 previously unclassifiable applications. In several instances, the reconstructed strings revealed sensitive runtime content, including cleartext credentials, encryption keys, and C2C endpoints, none of which appeared in the static code or resource sections.

Overall, the proposed string encoding and explainability framework extends the effectiveness of RAM forensics by introducing a semantically rich, interpretable layer that bridges low-level byte analysis with human-readable forensic evidence. By combining direct runtime string recovery, TF–IDF encoding, and token-level interpretability through LIME/SHAP, the approach not only achieves competitive detection accuracy but also enables the analyst to reconstruct the operational logic of malware in a transparent and forensically actionable manner. This integration of textual and spatial analysis represents a significant advancement over the current state-of-the-art, transforming volatile memory into a source of both quantitative and semantic IoCs with also a human-readable encoding.

7.4 Memory Analysis with Volatility Report

In this set of experiments, I analyse instead the full RAM of the acquired device according to the methodology in 6.2.2 (i.e. `AndroMemDump - Full RAM`) and shown in Figure 7.12, asking ***Can we interpret Volatility output in full RAM analysis?*** The full RAM analysis is forensically done with the tool `Volatility`, available for Python 2 and Python 3. Given an input file (i.e. the dump

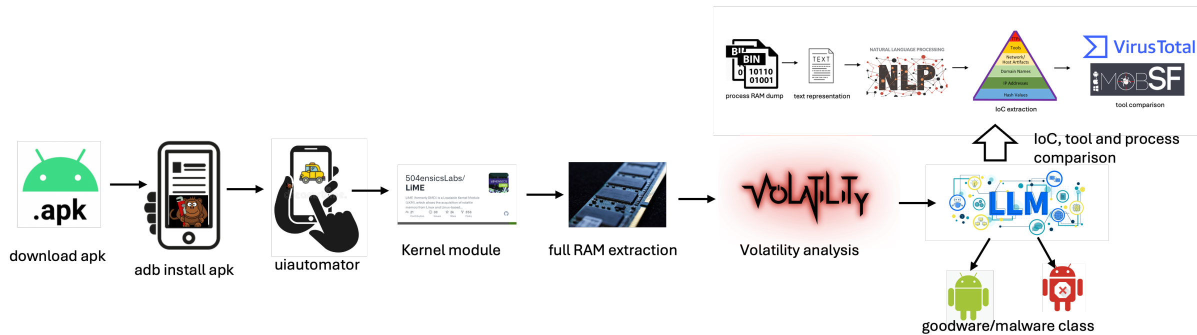

Figure 7.12: The picture shows the pipeline for analysing the full dump with Volatility and giving the output of the interesting plugins to a general-purpose LLM. In this way, the LLM can interpret the Volatility result and explain the content and why the dump contains specific found indicators. This analysis is compared with the one with Fridump text encoding

of the RAM), `Volatility` can extract a list of useful information such as process tree, connected IPs and domains, loaded libraries, list of files, etc. All these files are called plugins. As the Android device runs on Linux kernel, the tool `Volatility` needs to run the correct Linux plugin and selected the correct *profile*, i.e. the matching mapping table in order to retrieve the content from the memory as management by the correct running kernel version. The detailed procedure to have the correct profile analysis both for `Volatility` 2 and `Volatility` 3 is detailed in Section 6.2.2. Unfortunately, for Linux very few plugins have been released, especially if compared between version 2 and version 3, reflecting on the details of the results. Because of the issues about kernel recompilation, full RAM analysis is less explored even in the literature. The recent tool `VolMemDroid` [114] is one of the few. With respect to the methodology presented in Section 6.2.2, they follow the same pseudo-official methodology release from Volatility-Android [202], but for the analysis they use `awk` to extract specific IoC with regex and then convert them in two features (i.e. constant feature elimination and mutual information). On the contrary I analyse the output of `Volatility`, ask the general-purpose LLM for explanation and give back to the analyst a summarized content of the classification based on the most significant patterns, i.e. IoC, strings, `Volatility` plugins with interpretation.

This experiments have been conducted to understand the difference between the full RAM extraction and the methodology regarding the target process. The full RAM extraction requires more resources because of the kernel recompilation. On the other hand, the full RAM analysis gives more details not only targeted to the single process under analysis but also considering its effect in the whole system, for example monitoring written/accessed files, network connection, loaded libraries. For this reason, a list of `Volatility` plugins has been set (see Table 7.6 both for versions 2 and 3 and extracted their outputs). Each APK has been dumped 5 times during the first 60 seconds of execution (i.e. every 10 seconds which is also the time the used machine needs to dump the memory of 2GB, notably not corresponding to current real physical Android devices). Only a restricted subset of APK from the previous dataset has been considered, i.e. those in the category dropper and those fake APK (e.g. `Netflix, Twitter, Whatsapp, Instagram, TikTok, Facebook`) compared with the real official APK. After having saved and extracted every dump memory, each dump has been parsed with the Volatility plugins and saved the results for both versions. Now, I asked if general-purpose LLM such as `Chat-GPT`[1] and `Gemini`[2] could be used to understand the

[1] `https://chat.openai.com/`
[2] `https://gemini.google.com/`

Category	Volatility 2 Plugins	Volatility 3 Plugins
Process & Execution Analysis	linux_pslist, linux_psscan, linux_pstree, linux_paux, linux_psenv, linux_psxview, linux_threads	linux.pslist.PsList, linux.psscan.PsScan, linux.pstree.PsTree, linux.psaux.PsAux, linux.pscallstack.PsCallStack, linux.ptrace.Ptrace, linux.kthreads.Kthreads
Kernel & System Structures	linux_pidhashtable, linux_iomem, linux_lsmod, linux_ldrmodules, linux_library_list, linux_banner	linux.kallsyms.Kallsyms, linux.pidhashtable.PIDHashTable, linux.iomem.IOMem, linux.module_extract.ModuleExtract, linux.lsmod.Lsmod, linux.modxview.Modxview, linux.library_list.LibraryList
Malware & Rootkit Detection	linux_malfind, linux_hidden_modules, linux_check_creds, linux_check_idt, linux_check_syscall, linux_check_modules, linux_check_tty, linux_plthook	linux.malfind.Malfind, linux.hidden_modules.Hidden_modules, linux.check_creds.Check_creds, linux.check_idt.Check_idt, linux.check_syscall.Check_syscall, linux.check_modules.Check_modules, linux.check_afinfo.Check_afinfo, linux.netfilter.Netfilter, linux.keyboard_notifiers.Keyboard_notifiers, linux.ebpf.EBPF, linux.tty_check.tty_check
Filesystem & Page Cache	linux_enumerate_files, linux_recover_filesystem, linux_mount, linux_kernel_opened_files, linux_lsof	linux.pagecache.Files, linux.pagecache.InodePages, linux.pagecache.RecoverFs
Networking & Communication	linux_ifconfig, linux_netstat, linux_netscan, linux_route_cache, linux_sk_buff_cache	linux.ip.Addr, linux.ip.Link, linux.sockstat.Sockstat, linux.netfilter.Netfilter
Tracing & Instrumentation	linux_info_regs	linux.tracing.ftrace.CheckFtrace, linux.tracing.perf_events.PerfEvents, linux.tracing.tracepoints.CheckTracepoints
User Environment & Capabilities	linux_bash, linux_bash_env, linux_bash_hash, linux_dynamic_env	linux.bash.Bash, linux.capabilities.Capabilities, linux.envars.Envars
Miscellaneous & Device Interfaces	linux_truecrypt_passphrase, linux_dmesg	linux.kmsg.Kmsg, linux.graphics.fbdev.Fbdev, linux.vmaregexscan.VmaRegExScan, linux.vmayarascan.VmaYaraScan

Table 7.6: Comparison between Linux plugins in Volatility 2 and Volatility 3, grouped by functional typologies. The Volatility 3 list excludes redundant `linux.malware.*` duplicates, retaining only canonical plugin definitions.

`Volatility` output in order to help the analyst in understanding what is written but especially classify the dumps in goodware/malware by also giving an explanation on the string pattern and its filename that lead to such classification. This analysis is similar to the one based on strings for the target APK (see Section 7.3). For this reason, I adapted that methodology to this dataset and compared the results. Additionally, a custom NLP algorithm has been developed to parse both `Volatility` outputs but also the extracted strings both from the full RAM methodology (i.e. `LiMe`) and the target-process (i.e. `Fridump`).

The NLP component implements a rule-based, deterministic linguistic analysis framework for forensic text classification. Instead of using statistical or embedding-based models, it employs symbolic pattern recognition through cascaded regular-expression detectors that identify domain-specific cues such as command execution, network indicators, and privilege abuse. Each detected feature corresponds to a semantic frame representing a behavioral category (e.g. code loading, data exfiltration). Features are aggregated into a sparse, interpretable vector and evaluated using a linear scoring function. The score is then mapped to discrete categories (i.e. malicious, suspicious, benign, or unknown) according to fixed thresholds, as displayed in Figure **??**. Complementary entropy-based cues detect encoded or obfuscated content, enhancing sensitivity to concealment

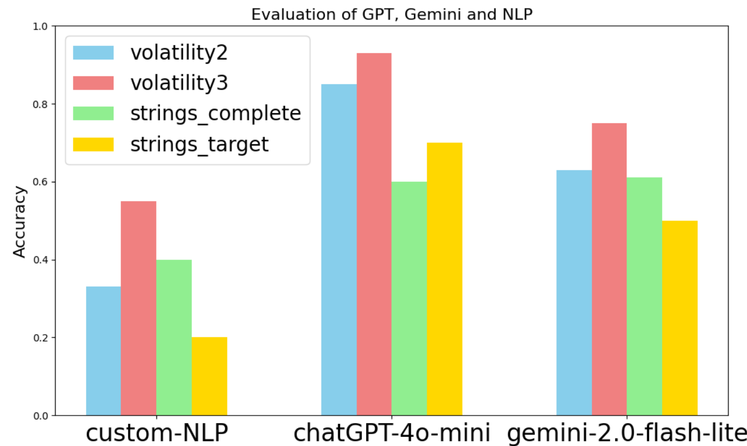

Figure 7.13: Comparison of the evaluation accuracy of the three models (from left to right: custom NLP, chatGPT-4o-mini, gemini-2.0-flash-lite) with the four different analysis (from left to right: volatility2, volatility3, strings extracted from the complete dump with LiME, and with the target dump with Frida

Fake APK	GPT Detection	Gemini Detection
Netflix	process name, malfind	process name, process entries
Twitter	connected IP, process name	process name, malfind
Tiktok	process name, hidden process, malfind	process name, hidden process, malfind
Facebook	process name, domain	manipulation, domain

Table 7.7: Comparison of GPT and Gemini string-based detections across a subset of fake applications

behaviors. The method thus constitutes a knowledge-driven, explainable NLP system, combining linguistic heuristics and information-theoretic metrics to provide transparent, reproducible forensic interpretations without relying on probabilistic language modeling.

First of all, I noticed that `chat-GPT` is strictly performant in analysing the output of `Volatility` and giving a human-readable description but also classifying them correctly. Moreover, they recognize the differences between the fake and real APK by also highlighting differences when asked. In fact, the fake APK can be easily detected because of the presence of different IoC regarding repacking, which is the technique to develop crack and fake APK starting from the existing one. Moreover, thanks to the plugin `malfind`, it can retrieve the injected malicious payloads, especially if not present in the real APK compared. In fact, both LLM, `chatGPT` and `Gemini` can identify the name of the package with malicious behavior (see Table 7.7).

LLM are more efficient than the custom NLP algorithm and this can depend because of the knowledge base of the general-purpose LLM. `Volatility` analysis with respect to the `Fridump` is more complete and LLM can better give explanations on the effects in the whole system. On the other hand, `Fridump` analysis allows to have a complete analysis, especially if done over time, on the behavior of the target application by understanding all its actions e.g. runtime permissions, API call, encryption, etc.

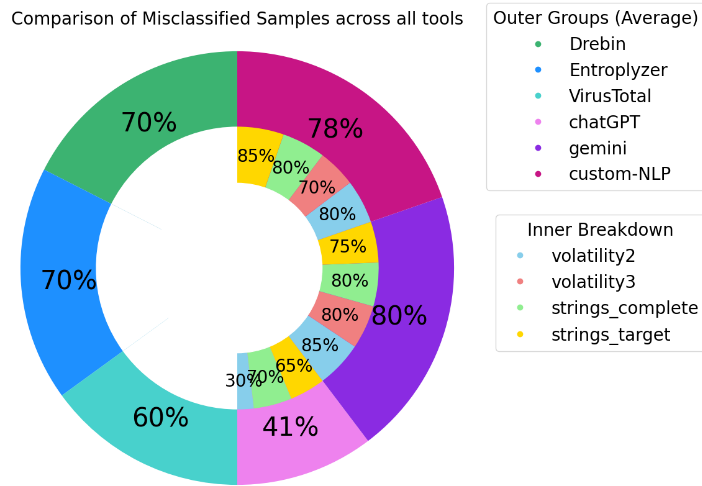

Figure 7.14: Average percentage of misclassified samples for the different detectors, i.e. Drebin, Entroplyzer, VirusTotal (notably average percentage of AV engines that fail to detect on VT) and the LLM-RAM approach with the different LLMs and RAM analysis methodologies (volatility2, volatility3, strings from the complete dump or the target process dump)

IoC / Evidence Category	VirusTotal	LLM–RAM (Memory Forensics)
URLs/IPs/Domains	✓	✓
Bundled files / archives	✓	✓
Files / artifacts on disk	✗	✓
Executable code / processes	✗	✓
System & kernel structures	✗	✓
User activity / environment	✗	✓
Malware / manipulation indicators	✗	✓
Device & memory interfaces	✗	✓
Tracing events	✗	✓

Table 7.8: Comparison of IoC categories supported by VirusTotal vs. LLM–RAM.

When comparing the extracted IoCs and strings from the memory dump with other state-of-the-art tools (i.e. `MobSF, VirusTotal, Drebin, Entroplyzer`), we can see from the results in Table 7.8 and Figure 7.13 about the misclassified samples, the importance of memory forensics analysis for malware detection. In this work, we highlight the interpretability of `Volatility` output thanks to the use of general-purpose LLM.

7.5 Detecting Android Stegomalware via Memory Forensics

Stegomalware refers to malicious software that conceals its payload or communication within seemingly benign carriers (e.g. images, audio, video, or network traffic) using steganographic techniques to evade detection. Unlike traditional obfuscation, which alters code visibility, stegomalware achieves stealth by embedding executable or command data within ordinary multimedia content, making it indistinguishable from legitimate data. The main research question of this work is ***Can we detect stegomalware via memory forensics?*** State-of-the-art detection methods focus on three complementary directions: *(i)* media-based forensics, which applies deep convolutional

or statistical steganalysis to detect hidden payloads in image, audio, or video carriers; *(ii)* behavioral and network-based detection, which identifies covert exfiltration patterns or anomalous communication timing; and *(iii)* hybrid AI-driven frameworks, which integrate media inspection with system-level monitoring through deep or graph-based models. Within the Android ecosystem, stegomalware is an emerging threat exploiting the platform's rich media environment and flexible resource model to embed or retrieve malicious code hidden in app assets, downloaded media, or shared data channels. Recent works have shown that adversaries increasingly leverage image-embedded payloads and social-media-hosted covert channels, motivating new multimodal detection pipelines that combine binary inspection, entropy analysis, and deep steganalysis for robust Android threat detection [55, 69, 188, 189].

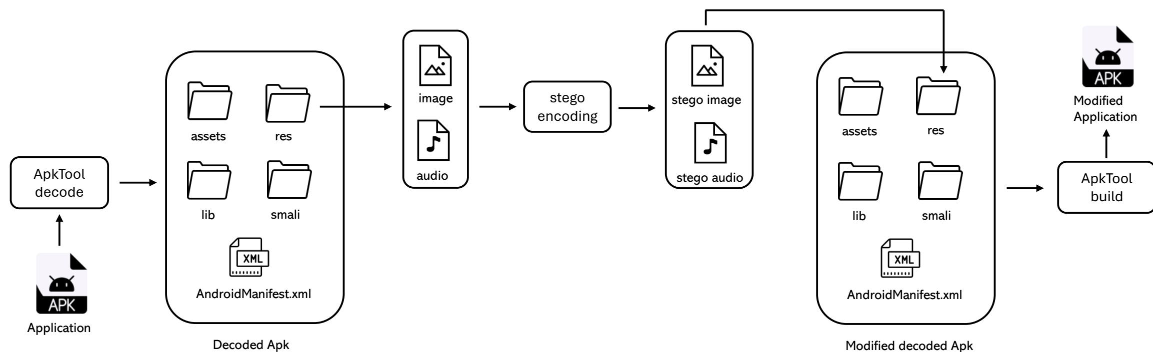

Figure 7.15: Workflow of the methodology employed to inject strings into APK raw resources and rebuild the application.

We examined in detail how modern Android malware can combine steganography with repackaging to avoid detection, and how the later phases of such attacks, in particular the loading stage, present unique challenges for analysis. The first part of this work concentrates on the feasibility and static detection. Given Android's market dominance and its role in critical workflows such as banking, authentication and personal communications, it is an attractive target for attackers. Although techniques such as code obfuscation and structural manipulation are well understood as evasion methods, the idea of hiding malicious payloads inside apparently benign media resources has received less attention, especially in the Android ecosystem.

To explore this, twenty of the most popular applications from the Google Play Store were downloaded and their internal images and audio files has been altered by embedding data with two least significant bit steganography schemes. The first, LSB plain, concealed information in the least significant bit of each color channel. The second, a higher-capacity method inspired by the OceanLotus APT, targeted three bits in red and green channels and two in the blue. For audio, the `AudioStego` [205] tool has been applied to embed short lists of malicious URLs. The payload was modeled on the `Necro` trojan [110], a malware that had recently compromised millions of devices. Up to two modified images and two audio files has been inserted per application, repackaged the software using `ApkTool`, and re-signed it with custom developer keys. Some high-profile applications, such as `Facebook` and `Instagram`, could not be rebuilt due to resource compilation errors caused by anti-tampering measures. Among the 67 applications that were successfully repackaged and remained functional, none were recognised as malicious by `VirusTotal`'s seventy-nine antivirus engines, aside from two false positives. Static detection mechanisms failed entirely to identify the hidden payloads,

revealing that no engines were parsing and inspecting media resources for steganographic content.

Application	Repacked	Detections	
		Original	Repacked
Amazon-Shopping	3	0	0
Bullet Echo	3	0	0
CandyCrush	5	0	1
Duolingo	15	0(*)	1
Eurospin	3	0	0
FlixBus	3	0	0
Telegram	35	0	0

Table 7.9: Repacked applications and their detection results by 79 VT antivirus. Repacked is the number of successful repacked applications, while Detections denotes the AV detections for the Original application and the Repacked one. The (*) symbol denotes that despite VT detection being 0, some malicious IoCs have been detected via static analysis.

In the second phase, the focus has been shifted on the carriers to examining the loading stage, i.e. the runtime code that identifies the hidden resource, extracts its contents, and executes it. In real-world stegomalware this component is as important as the steganographic media itself, yet it is often overlooked in detection research. Two versions of a loader has been implemented, shown in Figure 7.22: one that relied on Android's native `Bitmap` class to access RGB pixel data, and another using the `openCV` library to handle RGBA and LA images, loaded into the application context via the initDebug() function. Both versions located the image resource by its integer identifier, iterated through its pixels to recover the embedded bits, rebuilt the binary message, converted it to ASCII text, and executed it via Java/Kotlin reflection to avoid creating obvious static signatures.

Our threat scenarios considered two paths: in one, a legitimate application such as `Telegram` was modified to contain the steganographic loader and payload; in the other, a malicious application was built from scratch with these components included. In both cases the resulting APK could be distributed through alternative stores, sideloading, or by compromising the software supply chain. Obfuscated variants have also been created, using R8 for compile-time renaming and `obfuscapk` [33] for more extensive transformations such as control-flow reshaping, call indirection and reflection-based redirection, to examine how detection tools would respond.

The results showed that, whether in their original form or obfuscated, these applications were almost never classified as malicious by `VirusTotal`'s sixty-five Android engines. Dynamic sandboxes were sometimes triggered, but the tags they produced, such as "reflection" or "runtime-modules," were generic, appeared frequently in benign software, and offered no reliable basis for identification. Obfuscation altered outcomes inconsistently, at times slightly increasing detection rates but often leading to misclassifications without any clear indication of stegomalware.

Faced with the inability of static and conventional dynamic analysis to catch either the carriers or the loader, memory forensics has been investigated as a complementary detection method. The objective was to capture the hidden payload in memory after it had been decoded but before it was executed or freed. The methodology of target-process memory dumping has been applied in this scenario.

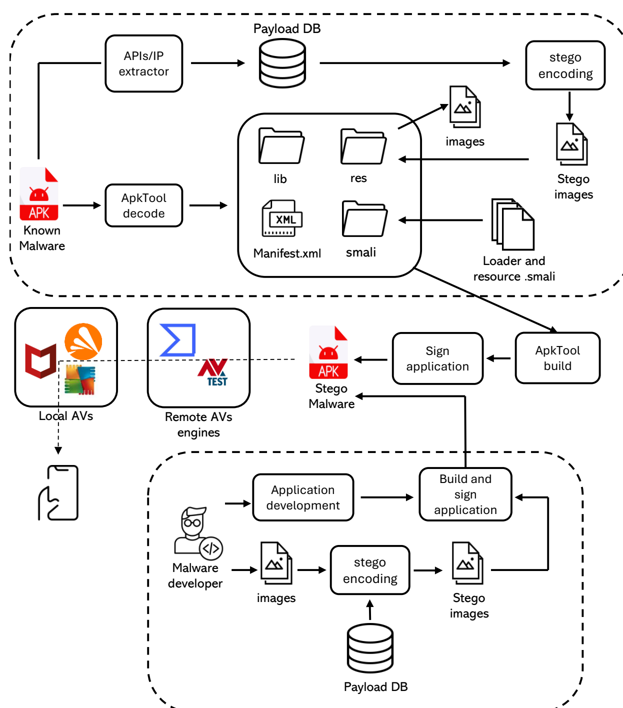

Figure 7.16: Upper attack pipeline: a known malware is repacked to include steganographic payloads, as well as the needed loading and execution functionalities. Lower attack pipeline: a brand new malware is written from scratch and endowed with the aforementioned functionalities.

The analysis of these memory dumps revealed the exact file paths of the image resources used as carriers, raw byte sequences of the extracted payloads, and the decoded payload strings in cleartext located in the Dalvik heap. This was possible because, once the loader finished its extraction routine, the payload existed as a contiguous sequence of bytes in a mutable object such as a StringBuilder before being handed to the reflection mechanism for execution. From the tests, these strings and byte arrays were found in the Dalvik Heap Main Space alongside other dynamically allocated objects, and their location within the dump made it straightforward to associate them with the loader's activity. The presence of the resource path gave a direct forensic link to the original embedded media file, while the raw byte data could be subjected to further analysis, for example by applying steganalysis algorithms in reverse to confirm the embedding pattern. Notably, the pure presence of steganography techniques does not mean the use of stegomalware as such techniques are used also as watermarking and copyright. Hence, it is important to analyse the content of the extracted payload. The cleartext strings, when extracted from the heap, could be examined for suspicious API calls, command-and-control URLs, IP addresses, or encoded commands. By correlating these three elements, shown in Figure 7.1 (i.e. resource path, raw payload, and decoded instructions) it is possible to reconstruct not only the fact that steganographic extraction had occurred, but also the intended behaviour of the malware.

This approach also allowed regex-based scanning of the memory snapshot for known malicious signatures or syntax elements, without needing to execute the payload in a sandbox. However, it is highly sensitive to timing: if the dump is taken too early, the payload has not yet been fully reconstructed; if too late, it may have been deallocated or overwritten by other processes, particularly if the loader employs active memory sanitisation techniques such as overwriting buffers or

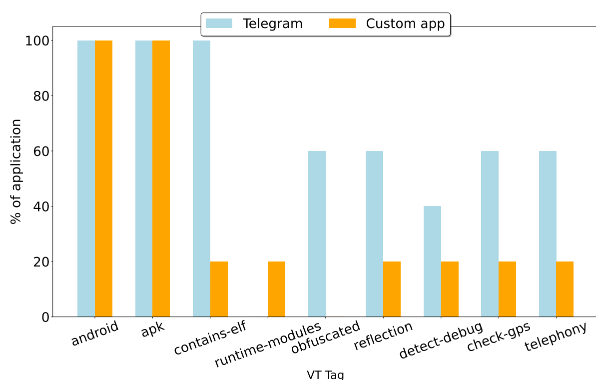

Figure 7.17: Percentage of APKs featuring the `Bitmap` loader variant for which VT marked with the specific tag.

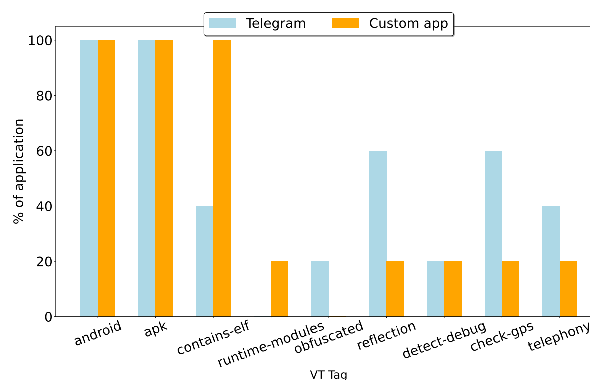

Figure 7.18: Percentage of APKs featuring the `openCV` loader variant for which VT marked with the specific tag.

Figure 7.19: Results comparison on VirusTotal tags with Bitmap (left) and openCV (right) implementation for the loader to extract the LSB steganographed content

Listing 7.1: Memory dump of the tampered Telegram application when data is hidden in a LA image, with the path of the resource, the sequence of bytes of the hidden payloads, and its cleartext.

```
...
0x3CF058: res/drawable-hdpi/fb_ic_checkmark_original_LA_seq_lsb.png
...
0x3ED9C8: [116, 101, 120, 116, 86, 105, 101, 119, 46, 115, ...]
...
0x3EDB54: textView.setText("This is a string set by an invocation taken from the image")
...
```

invoking the garbage collector immediately after execution. The method is also limited to payloads that are in plain text form at the time of inspection; if the malware uses an additional encryption or compression layer that is only decoded inside CPU registers or in ephemeral buffers, the payload may not appear in a usable state in the heap dump. Furthermore, non-rooted environments cannot attach `Frida` directly without embedding a `Frida-gadget` into the APK itself, which changes the binary and may not be feasible in some investigative contexts.

Despite these operational constraints, the memory forensics approach proved to be a reliable way of surfacing high-value indicators of compromise that were completely invisible to static and conventional dynamic scans. It demonstrated that a dedicated runtime memory inspection step could be integrated into the malware analysis pipeline to provide a more complete picture of multi-stage threats. The implications are that attackers can deliver operational stegomalware to Android devices without triggering current antivirus systems, that the loader stage is as invisible to default scanning as the steganographic carrier, and that memory forensics, although operationally more complex, can reveal forensic artifacts capable of linking the hidden data, its carrier, and the application's behaviour at runtime. A robust defensive posture against such threats will require the integration of static analysis of carriers, dynamic tracking of loader behaviour, and correlation with memory state to identify multi-stage attacks that rely on information hiding. Steganography can be used to hide specific significant features for the detection of specific malware families and in this way evade current ML detectors. Future studies will better address this topic.

These stegomalware works have been presented at ITASEC 2025 [189] and at IH&MMSec 2025

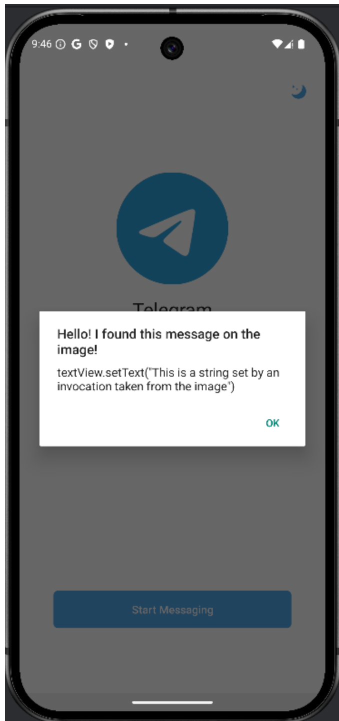

Figure 7.20: Tampered real application

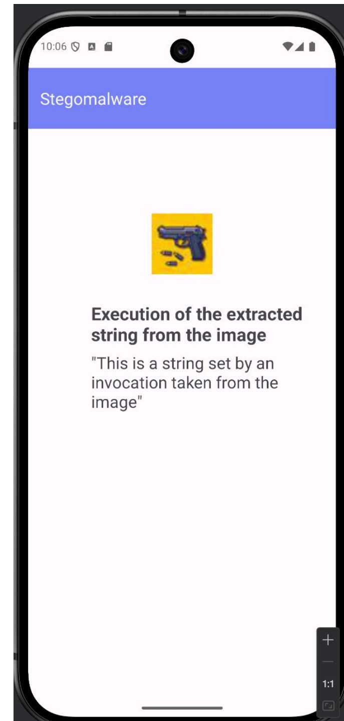

Figure 7.21: Custom tampered application

Figure 7.22: The hidden payload is extracted and executed on the two considered test applications.

[188], demonstrating how memory forensics can be a valid detection technique.

To conclude, by answering to the main research question that lead to this Chapter, the content of the RAM can be analysed into different encodings like visual images, audio, human-readable strings and interpretation of official tools, enabling the detection of stealthy malware, e.g. stegomalware, anti-analysis, obfuscated.

Approach	Research Question	Contributions	Results	Limitations
Image Encoding	How image encoding is valuable to detect malicious APKs from the memory dump?	Define a methodology to encode process RAM into 1D images compared to 2D	Detect anti-analysis malware with 95% accuracy; the data_stack is the most significant region; using Grad-CAM retrieve the most important bytes useful for human analysis	High computational cost, unfeasible in real-time monitoring
Audio Encoding	How RAM-based audio encoding is valuable to detect malicious APKs?	Evaluation of audio encodings for continuous signal memory-based malware detection	Wav8 is the best audio encoding, with accuracy 98%; the apk_full_stack is the most significant region.	Restricted dataset test
Text Encoding	How representative are the strings extracted from the RAM to detect malicious behaviors?	Comparison of text-based tools (VirusTotal and MobSF) with runtime strings in RAM	80% accuracy in malware detection; with SHAP and LIME extract IoCs not found by VirusTotal and MobSF	Restricted dataset on temporal tests
Volatility Report	How LLMs can interpret Volatility report and detect malicious in running process?	Comparison between extracted strings and Volatility report, and between general purpose LLMs and NLPs	ChatGPT-4o-mini and Gemini 2.0-flash-lite can successfully interpret the Volatility output and classify the app on the memory dump	Restricted dataset and tests on general purpose LLMs
Stegomalware Case Study	How feasible is the stegomalware in Android?	Feasibility study to apply stegomalware strategy in Android devices	Memory analysis can be used to detect the cleartext payload	No automatic detection pipeline

Table 7.10: Memory Analysis Contributions

Chapter 8

Applying Artificial Intelligence to Digital Forensics

In the previous Chapter, I described how memory forensics, a branch of DF, can be used in Android for malware analysis, detection, identification and explainability. During the works, a research questions arise automatically ***How to apply Artificial Intelligence to Digital Forensics?*** Hence, I introduced the use of AI in DF. However, memory forensics is not the only application scenario. In fact, I made a study on the use of AI in all DF phases. AI can be an assistant to the whole DF procedure but AI robustness must be considered to reduce the number of misclassification samples in real-world DF scenario analysis. Finally, I present how the DF procedure may change because of the creation of AI-generated content. In the near future many DF investigations will be based on the establishment of pure-generated content or real one. For example, one can intimidate another person but say that they did not do that, instead it is a AI-generated evidence. In this Chapter, I highlight the combination of AI in DF and viceversa.

8.1 Using AI in Digital Forensics

Recent DF tools, as discussed in Sections 1 and 4, are increasingly integrating AI to accelerate the identification and retrieval of specific media files without exhaustive device searches. The main research question of this work is ***To which other phases can AI be applied in DF?*** In the following I describe the study where we can adopt current AI techniques to all DF main phases.

For the *collection*, AI can help identify the digital devices to be seized at a crime scene. A camera with Computer Vision (CV) on object detection algorithms can highlight the digital devices to be seized by the first responder (i.e. the police and law enforcement appointed to access the crime scene and seize them). This will help to not forget to seize every digital device and follow the procedure by automatically taking pictures of the found devices, identifying patterns (e.g. if the device already has dents, scratches, or any damage) and reporting the data with an LLM in the chain of custody. I recommend using the CV model locally to prevent privacy without sending and collecting data on a server. Regarding *examination* where the memory (i.e. RAM and/or disk) of the seized digital device is acquired, AI can be applied as a digital assistant where the DF consultant writes the specifics of the case and the AI answers with a list of steps to be followed by using ad-hoc LLM. This is very important for the reconstruction of the chain of custody. Additionally, the model

is trained on different cases globally with anonymized data for better accuracy, considering only the technical procedure to guide the DF consultant. In this way, the acquisition of specific data would be facilitated by checking similarities with other cases. For example, it could help acquire corrupted memory sections or damaged disks, according to what has been done in similar cases in previous years. The *analysis* phase already uses AI algorithms to classify multimedia files. Still, such algorithms must be improved in terms of accuracy and robustness to presentation attacks. AI algorithms can also be trained on specific anti-forensics and anti-analysis techniques, such as steganography, to detect hidden patterns in files or memories. Training can be improved by using a shared global database of similar anonymized cases containing different techniques to analyze and retrieve readable data. In the *reporting* phase, the summary findings can be written in a draft using LLMs. For certainty, LLMs help in the background definition and glossary of the used terms. In the future, ad hoc trained LLM and NLP algorithms can help to read the document and assign a score based on DF understandability.

	Collection	**Examination**	**Analysis**	**Reporting**
AI	CV	LLM	ML & DL	LLM & NLP
Execution	Local	Local	Local	Local
Global Database	Seized Evidence	Data Acquisition	Similarities	Data Correlation
Privacy	Seized Scene	Personal Data	Irrelevant Data	Sensitive Information
Robustness	Adversarial Attacks	Anti-Forensics	Presentation Attacks	Hallucination
xAI	Seizure	Acquisition	Classification	Conclusions
Human Supervision	Object Identification	Data Integrity	Misclassification	Consistency

Table 8.1: Essential requirements for each DF phase. The table presents the use of specific techniques, security measures, and essential requirements for the collection, examination, analysis ,and reporting, and how to use them or for what specific action

Of course, these AI models help to improve and simplify the DF workload. It is important not to rely exclusively on them, and no DF consultant could report a result just because of the detection and classification of the AI system. Supervision is needed, and AI can be seen as an additional assistant. For a better justification of results, xAI must be used to understand why the model made that specific decision, improve the robustness, and improve classification.

While AI can significantly streamline DF work, it should be regarded as an auxiliary tool rather than a replacement for expert judgment. No conclusion should rely solely on automated detection or classification; human oversight remains essential. methods are crucial to clarify the reasoning behind AI decisions, improving robustness and interpretability. In this context, AI can serve as a digital assistant integrated into the DF pipeline, supporting pre-analysis phases where practitioners validate and interpret automated outcomes. Such AI-assisted consultancies could leverage recurring global case patterns to enhance detection and decision-making. In challenging

scenarios, for example those involving anti-forensics or steganography, pre-analysis AI modules can assist in locating and extracting concealed data. Training datasets can be augmented with evidence collected from Capture the Flag (CTF) challenges, validated by experts, to simulate anti-forensic cases, reflecting situations where practitioners must infer how a skilled adversary might conceal information. Moreover, generative AI can create synthetic yet realistic variants of such cases, enriching training data, improving feature extraction, clustering of evidence, and overall analysis efficiency even in the presence of advanced concealment techniques.

LLM such as `Gemini`, `Copilot`[1], and `ChatGPT` can be misused for illicit purposes, including circumventing detection systems, developing anti-forensic strategies, and acquiring technical skills like steganography. Although steganography itself is not inherently malicious (i.e. it is often employed legitimately to protect sensitive information) it is also a recognised technique for concealing malicious payloads. To investigate the practical capabilities of popular LLM in this domain, I conducted an experiment in which the three models were prompted to create an image containing hidden content using steganography, along with Python code to decode it. The instruction given to each model was: *"Can you generate an image with a hidden secret using steganography techniques? I also want the Python script to decode it."* All three models opted for the Least Significant Bit (LSB) approach, likely due to its simplicity and widespread use. In their initial outputs, none of the LLMs succeeded in producing a valid PNG image containing retrievable hidden data. Independent analysis with external tools, such as `zsteg`[2], `steghide`, and `stegcracker`, also failed to extract any embedded content from these generated images.

We then altered the experiment design. Instead of asking the LLM to produce both the image and the hidden content directly, supplying each with a plain white PNG image[3] and a message (*"This is a secret message APWG."*), requesting Python code to embed the message into the image using steganography and to produce a corresponding decoding script. Under these conditions, all three models generated functional scripts that could successfully hide and retrieve the secret message. Cross-testing confirmed that scripts from one LLM could decode messages embedded using scripts from another. Despite this, when the steganographed images were provided back to the LLM for decoding, none were able to extract the hidden content, even from images generated using their own code. The scripts themselves varied in structure and style, and only `Copilot` required an explicit clarification that the task was for educational purposes to avoid triggering content safety restrictions. The `ChatGPT` decoding script required minor debugging due to an error in pixel index calculations, but all three models' scripts worked correctly once fixed. For further validation, the generated decoding scripts were tested on an external benchmark dataset[4][5] containing images with known LSB-based hidden content. In all cases, the scripts successfully extracted the embedded strings.

The results, summarised in Table 8.2, indicate that while LLMs currently lack the capability to directly process and decode steganographed PNG images in a conversational setting, they can produce functioning encoding and decoding scripts (often correctly on the first attempt) that work reliably on both their own generated outputs and on real-world datasets. This work has been

[1] `https://copilot.microsoft.com/`
[2] `https://github.com/zed-0xff/zsteg`
[3] `https://upload.wikimedia.org/wikipedia/commons/d/d2/Solid_white.png?20060513000852`
[4] `https://github.com/Ocram95/AndroidStego/tree/main/resources/stego_resources/LSB/Sequential`
[5] `https://github.com/Diesoi/AndoidStegoLoader/tree/main`

Input	Encoded String	Decoding Scripts				Chat Loading	Interaction
		Gemini	Copilot	GPT	Zsteg		
Gemini Generated	–	✗	✗	✗	✗	✗	–
Gemini Script	This is a secret message APWG.	✓	✓	✓	✓	✗	1
Copilot Generated	This is a secret!	✓	✓	✓	✓	✗	–
Copilot Script	This is a secret message APWG.	✓	✓	✓	✓	✗	1
GPT Generated	The password is swordfish	✗	✗	✗	✗	✗	–
GPT Script	This is a secret message APWG.	✓	✓	✓	✓	✗	2
GitHub Dataset	..."rrqnDG4dja7Ga5ZdAuD77CY" `textView.setText("string_here")`	✓	✓	✓	✓	✗	–

Table 8.2: Decoding results for various AI-generated, scripted, and real steganographic images.

presented at APWG Tech Summit 2025 [178] and describes how the current AI techniques can be applied to DF phases.

8.2 Evaluating the Robustness of AI-based Digital Forensics Tools

The growing integration of AI into DF tools has profoundly reshaped investigative workflows, automating the identification, analysis, classification, and triage of vast quantities of multimedia evidence. While these AI-enhanced systems promise efficiency and relief from the cognitive and emotional burden traditionally placed on human analysts, their increasing autonomy introduces new risks concerning robustness, transparency, and evidential reliability. The main research question of this work is ***How robust are DF tools using AI?*** In particular, the proliferation of AI-generated or manipulated media (e.g. deepfakes, morphed images, and adversarially perturbed content) poses a critical challenge to the integrity of automated forensic analysis. If a forensic classifier erroneously labels falsified or altered data as genuine, or fails to detect relevant but atypical evidence, the entire investigative process may be compromised. Despite these risks, few independent studies have systematically examined how commercial forensic suites behave under adversarial or deceptive conditions. Our research addresses this gap by conducting a two-stage evaluation. First, a broad assessment of AI-driven forensic tools under realistic, potentially anti-forensic scenarios. Subsequently, a controlled and quantitative investigation focused on their robustness against facial manipulation attacks.

In the first assessment on AI-driven forensics tools, the integration of AI within commercial DF software has been examined by addressing the influence on the reliability and interpretability of automated evidence classification. The increasing adoption of AI modules in forensic workflows (e.g. those embedded in `Magnet AI` by `Magnet Axiom` and `Excire Photo AI` by `X-Ways Forensics`) raises critical questions about their robustness in the presence of adversarial manipulations, contextual variability, and anti-forensic practices. Since these black-box systems are often employed to pre-filter massive volumes of multimedia evidence, any misclassification or omission may directly affect the outcome of an investigation. The selected tools can recognize nudes, drugs, weapon from images and chats. To investigate these risks, I designed a preliminary, scenario-oriented experimental framework focused on the empirical behaviour of such tools under realistic and potentially adversarial conditions. Given the lack of publicly available datasets suitable for this purpose, I constructed ad-hoc data emulating common categories encountered in DF investigations. The restricted dataset

included 200 images for nudity classification spanning real and stylized pornographic content, drawings and manga scenes, and clothing printed with nude bodies. Additionally, I considered 200 facial images representing genuine and AI-generated portraits of well-known individuals (e.g. actors Brad Pitt and Angelina Jolie) and their deepfakes, also considering pictures of their child to evaluate parental similarities, as well as subjects before and after cosmetic or gender-affirming surgery. To study the chat efficiency, I created three `Facebook Messenger` restricted chat datasets, each exceeding one hundred messages and attachments, thematically focused on sex, drugs, and weapons, and deliberately written using both explicit and coded language (i.e. synonyms or gibberish). These data were processed through `Magnet Acquire` and analyzed in `Magnet Examine` via the *"Analyze Pictures/Chats with AI "* functions, while `Excire Photo AI` was queried using textual prompts and facial similarity searches. We evaluated the correctness and consistency of the tools' automated categorizations across multiple machines, recording instances of false positives, missed detections, and unstable classifications. This first study was conceived as an exploratory validation effort, prioritizing ecological validity and operational insight over quantitative rigor, in order to highlight how commercial AI-driven forensic tools respond to non-standard, adversarial, or ambiguous inputs.

Building on the findings of this preliminary investigation, a subsequent and more targeted study has been conducted, evaluating forensics face recognition on deepfakes and morphs, performing a deeper and quantitatively grounded analysis of the same AI-based tools against facial manipulation attacks. In this second phase, I focused on the specific vulnerabilities of `Magnet AI` and `Excire Photo AI` when handling synthetic biometric data, such as face-swapped videos or morphed identities, which represent some of the most critical challenges for digital and biometric forensics. To ensure reproducibility and comparability, two state-of-the-art benchmark datasets has been adopted: `Celeb-DF` for deepfake imagery and `CelebAMask-HQ` for morphing. For the deepfake and morphing evaluation, a rigorous dataset preparation process has been adopted to ensure both forensic relevance and experimental reproducibility. Deepfake samples were sourced from the `Celeb-DF` benchmark, a large-scale dataset comprising 590 authentic and $5,639$ manipulated celebrity videos generated through advanced face-swapping techniques. Since `Excire Photo AI` performs image-level rather than video analysis, the first frame from each video has been extracted using `FFmpeg`, preserving the original frame rate, resolution, and color profile to prevent the introduction of processing artifacts that could bias the AI classifiers. Each frame retained its original identity label, allowing precise correspondence between genuine and synthetic representations of the same subject. For the morphing experiments, the `CelebAMask-HQ` dataset has been employed, which contains $30,000$ high-resolution celebrity portraits curated from the `CelebA` dataset. From this dataset, 92 frontal, neutral, and unobstructed images has been manually selected, ensuring homogeneous lighting and the absence of occlusions or accessories that might interfere with facial landmark detection. Based on these curated samples, $5,108$ morphed images were generated using a landmark-based geometric blending approach implemented through a modified version of `Face Morpher`. The procedure involved detecting facial landmarks, applying Delaunay triangulation for spatial alignment, and performing weighted pixel blending (0.5 ratio) between the source and target faces. To mitigate artifacts such as ghosting or boundary misalignment, the morphing was restricted to the inner facial region, excluding background and hair contours. Each image pair produced two complementary morphs, denoted M_{12} and M_{21}, representing both transformation directions. All resulting images were stored in their native dimensions and color space without compression or resizing to pre-

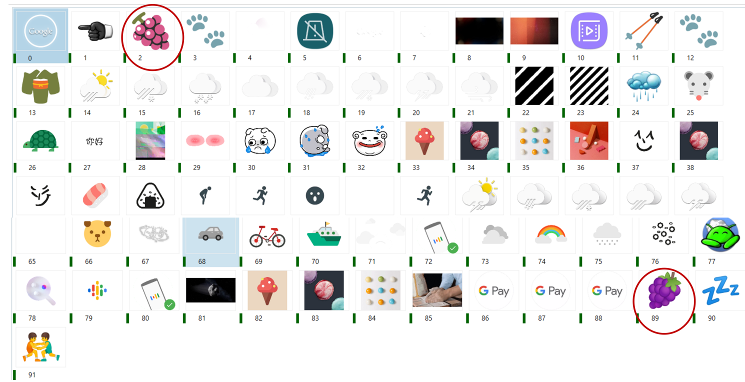

Figure 8.1: The image shows the pictures detected as drugs but that clearly are not drugs (false positive rates) such as the grape (red circled) for the drug detection in chat done by Magnet AI tool.

serve fine-grained visual details relevant to authenticity analysis. This systematic data preparation yielded a controlled yet diverse distribution of bona fide, deepfake, and morphed images, providing a consistent foundation for assessing the robustness of AI-driven forensic tools under standardized and reproducible conditions.

Through this two-stage methodology I established an empirical baseline for understanding the limitations and operational risks of current AI integrations in digital forensics. The first study provided qualitative insights into real-world reliability and anti-forensic vulnerabilities, while the second delivered a standardized and quantitative assessment of algorithmic resilience to facial manipulation, thereby completing a comprehensive methodological investigation into AI robustness in forensic practice.

The results of the first experimental assessment revealed that the integration of AI-based classifiers within forensic suites still lacks the robustness required for dependable evidential interpretation. When evaluating `Magnet AI`, I observed that the system performed inconsistently across different types of visual content and chat scenarios. In the nudity detection experiments, the classifier correctly recognized the majority of genuine pornographic and adult images, even across diverse skin tones and lighting conditions, yet it frequently failed to detect stylized or ambiguous representations of nudity. For instance, images depicting painted dresses or manga-style drawings were often mislabeled as non-explicit, despite their clear semantic association with nudity. Conversely, certain benign images containing body-like textures or clothing patterns were incorrectly categorized as explicit content, revealing a lack of semantic contextualization within the classification model. Quantitatively, the detection rate for real pornography was the highest among the categories tested, whereas the t-shirt and illustrated datasets exhibited significant false negatives, demonstrating that `Magnet AI` relies heavily on low-level color and texture cues (e.g. skin-tone distribution) rather than on deeper contextual or structural understanding of human anatomy.

The chat analysis component of `Magnet AI` produced even more limited results. Across the three simulated `Facebook Messenger` conversations (focused respectively on sex, drugs, and weapons) the AI successfully identified explicit visual content only in a subset of transmitted images, while it failed to correctly classify or tag the textual messages themselves. Neither explicit nor metaphorical references in the chat text (e.g. slang expressions like “snow” or “white powder” for cocaine) were detected, indicating that the underlying natural language processing model lacks domain adaptation for contextual inference. Moreover, some unrelated default images retrieved during the device dump

(e.g. stickers or pre-installed icons) were falsely classified as drug-related, producing false positives unrelated to the evidence set, as shown in Figure 8.1. These findings suggest that the AI classifier embedded in `Magnet AI` has not been optimized for real-world conversational variability, resulting in high sensitivity to visual cues but near-zero understanding of linguistic context.

The face recognition tests performed with `Excire Photo AI` similarly exposed the fragility of black-box AI algorithms when applied to forensic image analysis. When querying the system with celebrity names such as Brad Pitt and Angelina Jolie, the tool successfully retrieved a portion of genuine photographs but also returned a substantial number of false positives, including deepfakes and unrelated faces that visually resembled the queried individual. For the Brad Pitt dataset, for example, only three of twenty authentic images were correctly identified, while several deepfake images of the same person were accepted as genuine. The system also demonstrated instability in handling familial similarities: when searching for Shiloh Jolie-Pitt, the results included deepfakes of Brad Pitt and unrelated faces labeled as matches, confirming that the algorithm's internal representation overemphasizes surface similarity rather than biometric distinctiveness. Similar issues arose in the cosmetic and gender-affirming surgery tests, where some subjects (e.g. Donatella Versace) were no longer recognized after surgery, while others (e.g. Sylvester Stallone, Cher, Courtney Cox) were inconsistently matched with additional unrelated individuals. The false match rate was therefore non-negligible, and in several cases the classifier's visual similarity retrieval produced implausible associations, such as matching different celebrities or unrelated deepfakes under the same identity.

Overall, the results demonstrate that both AI modules, though functional in benign scenarios, lack the semantic robustness and adversarial resistance necessary for forensic reliability. `Magnet AI` shows acceptable performance for straightforward image-based classifications but collapses when confronted with artistic, ambiguous, or adversarial examples, while `Excire Photo AI` fails to maintain discriminative precision under even moderate manipulations such as deepfakes or post-surgical facial changes. Moreover, both systems exhibit limited transparency, neither provides any confidence score or interpretability mechanism to justify a classification outcome, thereby preventing forensic practitioners from validating the credibility of automated results. These weaknesses collectively emphasize that AI-assisted tools, when employed in forensic contexts, must be treated as auxiliary decision-support systems rather than autonomous analytical agents. Without explicit validation, explainable outputs, and adversarial testing, their classifications remain prone to systematic error, potentially leading to misleading evidence interpretation in investigative and judicial processes.

The evaluation conducted on the `Celeb-DF` and `CelebAMask-HQ` datasets provided a detailed quantitative understanding of how current AI-based forensic tools behave when compared with realistic facial manipulations. In the face detection stage performed through `Magnet AI`, the system exhibited robust baseline performance, correctly identifying facial regions in both authentic and forged images. The False Negative Rate (FNR) remained low (i.e. 0.34% for real videos and 0.29% for deepfakes in `Celeb-DF`, and 0.00% for both real and morphed faces in `CelebAMask-HQ`, confirming that the presence of facial features was consistently detected regardless of manipulation. Such results are report in Table **??**. However, when the same system was tasked with distinguishing AI-generated content from authentic images, the results deteriorated drastically. The average FNR, reported in Table 8.3 exceeded 99.8%, and the Balanced Accuracy dropped to approximately 49˘50% across both datasets, indicating that `Magnet AI` could not effectively differentiate real from synthetic images.

FNR [%]			
Celeb-DF		**CelebAMask-HQ**	
Real	Deepfakes	Real	Morphs
0.34	0.29	0.00	0.00

	FNR [%]	**FPR [%]**	**Balanced Accuracy [%]**
Celeb-DF	99.85	0.51	49.81
CelebAMask-HQ	99.90	1.09	49.51

Table 8.3: Results of the detection of facial manipulations through Magnet AI. Deepfakes and morphed images are considered the positive class in Celeb-DF and CelebAMask-HQ, respectively.

In other words, the probability that a manipulated image would be recognized as AI-generated was virtually zero, suggesting the model relies on superficial heuristics (e.g. texture or luminance patterns) rather than robust forgery-aware representations. The False Positive Rate (FPR) for unaltered samples was below 1.1%, showing that genuine content was almost never mislabeled as fake, further highlighting the system's extreme bias toward false negatives, a critical vulnerability in forensic verification scenarios.

The identity recognition, reported in Table 8.4 analysis performed with `Excire Photo AI` yielded comparable outcomes but with a more complex behavioral pattern. Under normal conditions, when querying authentic images of known individuals, the tool reached balanced accuracies between 86% and 91%, with FPR values around 2˘9%, depending on the dataset and the diversity of the imagery. When confronted with zero-effort impostor attacks (images of unrelated identities), the recognition accuracy remained relatively stable, indicating the tool's competence in conventional recognition tasks. However, the introduction of facial manipulations, whether through deepfake or morphing processes, produced a severe degradation in reliability. In the `Celeb-DF` dataset, `Excire Photo AI` exhibited an False Match Rates (FMR) of 68.1% for deepfakes, while in `CelebAMask-HQ`, morphing attacks raised the FMR to 75.0%, with False Non-Match Rates (FNMR) of approximately 40% in both cases. This means that manipulated faces were frequently accepted as genuine matches for the searched identity, especially when the target identity was used as the source in the forgery process, as shown in Figure 8.2. Conversely, images in which the target identity appeared as the recipient (i.e. as the morphing target) were more often rejected, suggesting that the AI relies primarily on the dominant facial structure or texture cues contributed by the source image rather than on stable biometric characteristics. Further qualitative inspection confirmed that `Excire Photo AI` recognition decisions were inconsistent and often accompanied by visually implausible matches, such as misidentifying morphs of two unrelated celebrities as valid representations of a queried subject. In several cases, the tool incorrectly linked manipulated images to both contributing identities, indicating that its feature embeddings overlap across forged and genuine samples. This confusion was especially evident when morphing was performed between individuals sharing similar facial geometry, supporting the hypothesis that the model prioritizes low-level shape similarity rather than identity-specific discriminative features.

These findings reveal an acute susceptibility to presentation attacks, where altered or synthetic

	Impostors compared with genuine data	FNR [%]	FPR [%]	Balanced Accuracy [%]
Celeb-DF	**Zero-effort impostors**	24.73	2.29	86.49
	Deepfakes		39.09	68.09
CelebAMask-HQ	**Zero-effort impostors**	9.78	8.83	90.69
	Morphed images		40.21	75.01

Table 8.4: Genuine original (unmanipulated) images presenting the searched identity are considered to belong to the positive class.

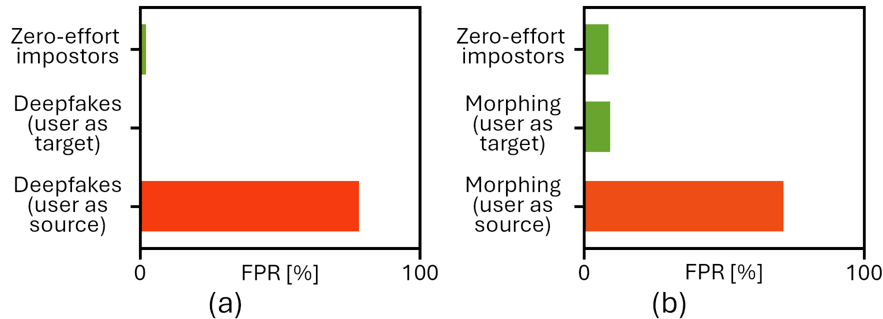

Figure 8.2: FPR obtained in face recognition through `Excire Photo AI` when comparing genuine original images (positive class) with zero-effort impostors, deepfake images (a), and morphed images (b).

faces bypass the AI classifier's internal decision boundaries, leading to false attributions that could compromise evidential validity in forensic investigations. When comparing the two tools, `Magnet AI` demonstrated strong face detection robustness but almost total failure in authenticity discrimination, while `Excire Photo AI` retained limited recognition accuracy but was highly vulnerable to identity-based deception. Both tools performed adequately under benign conditions yet collapsed under manipulative stress, confirming that their internal AI subsystems, though effective for semantic labeling and retrieval, are not designed to withstand adversarially generated inputs. This is particularly concerning for digital forensics, where manipulated content can be introduced intentionally as part of anti-forensic strategies. Overall, these results underscore a systemic limitation in current AI-enhanced forensic pipelines: they can automate evidence discovery but cannot guarantee the authenticity or identity integrity of analyzed visual data. The combination of extremely high false negatives for deepfake detection and high false positives in facial recognition compromises the trustworthiness of AI-based categorization in legal or investigative contexts. These findings suggest the urgent need for developing forensic tools that integrate forgery-aware training, xAI mechanisms, and multi-modal verification strategies, capable not only of identifying content but also of quantifying its authenticity with verifiable confidence scores. A pre-print of the work can be found on arXiv [177], while the evaluation part of deepfakes and morphing has been presented at EUSIPCO 2025 [175], demonstrating that current tools are still not robust and mature to be used without any human supervision.

8.3 Digital Forensics Analysis of AI-generated content

The increasing use of genAI and the easiness of generating synthetic and false data can also affect the DF world, automatically asking the research question ***What is the future of DF with gen-AI***

content? Many tools have been developed and freely released to mimic a real person but in a fake scenario (e.g. deepfakes) or ultimately generate new data from nonexistent content (e.g. complete synthetic data). To the best of my knowledge, based on my personal experience in DF consultancies and feedback from other local, national, and international consultants and law enforcement, people are starting to declare in a trial that specific multimedia files (i.e. audio and video) were generated with AI and they did not say or made that specific action. Hence, the main DF questions for future research on genAI data will be *How to prove that the data is generated by AI mimicking the real person?* Sometimes, the human eye can recognize fake data due to recognizable artifacts in videos or pictures depicting humans. For example, such pictures may contain incoherent lights and shadows, human hands with an incorrect number of fingers, or unrealistic gestures. Another interesting future research question regards the attribution of the created genAI media, hence *How to prove who made the synthetic audio/video/image?* This is specifically for complaint cases where fake media is generated to defame a person. This would be the future of DF, and many techniques must be developed accordingly. These questions could be solved with the use of steganography, for example, by adding a watermark in the file or artifacts in the metadata to trace signatures from the tool used. Some companies cannot agree on these specifics. Hence, there should be regulated standards as in the European AI Act[6]. Another strategy for detecting genAI multimedia files would be a deep file structure analysis. We all know that the structure of images, audio and video follow specific patterns, e.g. pixels have specific structures, audio have specific frequencies, video have specific time sequences. In fact, deepfakes do not always have human natural and biological signs. Recurrent unrealistic patterns, such as those highlighted previously, can be detected with a deep analysis. These patterns can also be used as features for training specific AI algorithms. Such features can also be established by a deep and detailed comparison of similar real data, by finding the differences, or by association with other similarly generated data.

This work has been presented at APWG Tech Summit 2025 [178] and presents some plausible solutions based on standards on how to forensically determine the nature of a genAI file.

To conclude, by answering to the main research question that lead to this Chapter, the AI can be applied to all DF phases, not only for automatic data classification during analysis. Still, some improvements about robustness must be made to have more accurate tools, considering for example large datasets, representative of reality and data manipulations. About the latter, specific standards must be adopted in order to forensically determine if a file is genuine or produced by genAI algorithms.

[6] `https://digital-strategy.ec.europa.eu/en/policies/regulatory-framework-ai`

Approach	Research Question	Contributions	Results	Limitations
AI in DF	Is it possible to apply AI in all DF phases?	Use current AI-based techniques on collection, examination, analysis and reporting	Use general purpose LLMs as DF assistant for analysis	No tests on real dataset
Robustness of AI-based DF tools	How robust are the AI algorithms in DF tools for automatic analysis and classification?	Tests on AI robustness and image manipulations applied to DF tools	Some categories $<20\%$ accuracy	Restricted dataset and manipulations
DF analysis of genAI	What is the future of DF investigations with genAI multimedia files?	Propose a standard to recognize genAI multimedia files	Steganography and watermarking for copyright	No tests on real dataset

Table 8.5: Artificial Intelligence and Digital Forensics Contributions

Chapter 9

Conclusions

In this thesis, I explored the attack flow from native code vulnerabilities in Android applications that, if exploited, can lead directly to the main memory. Native code is the set of C/C++ libraries and Java Native Interface libraries that allow access to the native activities and components in an Android application, such as camera interaction, image processing, and fast data processing. The C/C++ code can be directly implemented by the developer or imported by third parties. In both cases, if some inputs, conditions, data structures, and code flows are not managed correctly, typical C/C++ vulnerabilities can emerge, such as buffer overflows and format string vulnerabilities. Each vulnerability, once discovered, is publicly released in the CVE format. Nowadays, many popular APKs, such as social networks, contain known CVEs that are not patched, which, if exploited, can lead the user to risky attacks [179]. To be exploited, a vulnerable function must be called from the Java code of the APK. This analysis is called reachability and refers to the construction of a graph from a Java function to a vulnerable C/C++ function, through the library that interacts with the Java code and the native code. Once the vulnerable function is reachable, the correct input that triggers this vulnerability must be identified. These techniques are called fuzzing. AI-based algorithms are fundamental in reachability and fuzzing tasks because they allow fast data processing, data classification, retrieving the best vulnerable path, and the context-aware input to exploit the target vulnerability.

Once the vulnerability is triggered in the native code, the attacker has access to the main memory. In RAM, all data is stored temporarily in clear text; in fact, it is possible to retrieve specific secrets such as one-shot pictures, encryption keys, and temporary data. For this reason, I used memory forensics techniques to detect Android malware. In particular, I focused on specific families such as trojans, droppers, fake apps, stegomalware [188, 189]. Two different memory forensics acquisition strategies has been adopted: one based on the complete device RAM analysis and another based on the RAM allocated to the target process. Subsequently, various data encoding techniques has been applied for memory analysis and elaboration, including images (2D and 1D), audio, text, and the output of common tools. The encoded memory data has been passed to various AI algorithms, including CNNs, LLM models, and NLP algorithms, to quickly classify the APK and identify the most important memory regions and bytes that contribute to this classification. Thanks to such encodings and AI-based techniques, it is possible to identify new IoCs beyond state-of-the-art tools and detect APKs with anti-analysis or evasion behaviors beyond other currently available tools.

This malware analysis approach can be easily adapted to real Android devices, not only emulated ones. For both analyses, it is necessary to acquire root (e.g. super-user, administrative privileges) and sometimes recompile the kernel. Such procedures can be challenging, but I successfully constructed an efficient pipeline. In order to make this procedure forensically aligned on real devices, while acquiring super-user privileges for the resolution of a juridical case (e.g. a cyber incident or a traditional process where an Android device can contain important data), it is necessary not to delete data. Otherwise, the evidence is compromised and the data is lost. In this way, MoLIFE [173] has been developed, an Intelligent-based DF monitoring and acquisition process based on the concept of Digital Twin, adapted to mobile devices. Hence, the setup results in an emulator with root privileges, fully synchronized in real time with the physical device (mobile Digital Twin – mDT), which itself does not possess root privileges. Specific monitoring and data acquisition can be assessed in the mDT as if operating on the real physical device. This methodology can also be applied after an incident, provided that the device owner collaborates in unlocking the device.

AI algorithms can also be applied to each DF phase, from the collection to the reporting [178]. For example, Computer Vision techniques can be applied in the evidence collection. AI-based analysis tools must be both accurate and robust the minimize misclassification errors, while also being resilient to specific presentation attacks, such as deepfakes, morphing, or data hidden with steganographic techniques. AI-based analysis tools are extremely powerful tools that allow a fast elaboration of a huge amount of data i.e., considering not only the possible terabytes to be analyzed in case of an incident to companies, but also taking into account the different number of cases that must be analyzed. Moreover, such algorithms are fundamental for the user privacy preservation but also to protect the analyst from psychological shocks (e.g. when analyzing a huge amount of pedo-pornography data). Regarding presentation attacks [175, 176] such as deepfakes and morphs, the future of Digital Forensics investigations should be more focused on the detection of gen-AI content, specifically on the attribution of multimedia file (e.g. representing people, politicians, nudity, etc) to real world (i.e. a real taken picture) or to gen-AI content (i.e. a deepfake).

Despite the promising results, this thesis still has some limitations. Due to the lack of proper, real, and publicly available datasets, some methodologies may have been tested only partially or with a limited subset of data. First of all, the vulnerability tests must be executed in a real-world scenario, targeting impact on APKs. The risk methodology can be applied to vulnerabilities in the Java code, hence analyzing the APK in its entirety. Still, many research questions can be answered in future works. One example is the robustness evaluation to adversarial attack in malware detection systems based on memory forensics features. Moreover, other Digital Forensics extracted data can be used for the attack detection, as highlighted in MoLIFE, but still not tested. In particular, the most interesting tests concern the application of AI in DF investigations, which require a proper real dataset with the collaboration of the companies involved. Finally, it is necessary to identify an alternative solution to the rooting problem that does not rely on the use of the mDT. Although the mDT can represent an effective approach, it requires a considerable amount of server capacity, computational resources, and energy consumption when applied to a large number of potential targets (e.g., critical infrastructures, sensitive companies, or individuals at risk).

Additionally, as Hu *et al.* [104] conducted a study proving that the majority of techniques can be easily adapted to iOS, it should be very interesting to apply this thesis on iOS devices in order to compare the two mobile OS worldwide.

Bibliography

[1] android-afl. Available online: https://github.com/ele7enxxh/andr oid-afl/blob/master/README.md.

[2] Android greybox fuzzing with afl++ frida mode. Available online: https://blog.quarkslab.com/android-greybox-fuzzing-with-afl-frida-mode.html.

[3] Android ndk. Online accessed: https://developer.android.com/ndk/guides.

[4] Android operating system. Online accessed: https://source.android.com/.

[5] Android platform architecture. Online accessed: https://developer.android.com/guide/topics/manifest/ma intro.

[6] Common vulnerability and exposure (cve). Online accessed: https://www.cve.org/About/Overview.

[7] Common vulnerability scoring system (cvss). Online accessed: https://www.first.org/cvss/v3.1/user-guide.

[8] Fuzzing with libfuzzer. Available online: https://source.android.com/docs/secur ity/test/libfuzzer.

[9] Iso/iec 27043:2015 – information technology – security techniques – incident investigation principles and processes.

[10] Librarian github repository. Available online: https://github.com/salmanee/Librarian.

[11] National vulnerability database from nist. Online accessed: https://nvd.nist.gov/.

[12] Pwntools. Online accessed: http://docs.pwntools.com/en/latest/.

[13] Repacking ios applications. https://labs.withsecure.com/publications/repacking-and-resigning-ios-applications.

[14] Iso/iec 27037:2012 – information technology – security techniques – guidelines for identification, collection, acquisition and preservation of digital evidence, 2012.

[15] Iso/iec 27041:2015 – information technology – security techniques – guidance on assuring suitability and adequacy of incident investigative methods, 2015.

[16] Iso/iec 27042:2015 – information technology – security techniques – guidelines for the analysis and interpretation of digital evidence, 2015.

[17] Iso/iec 27005:2018 – information technology – security techniques – information security risk management, 2018. Defines a framework for information security risk management within the ISO/IEC 27000 family of standards.

[18] Androguard VirusTotal. `https://docs.virustotal.com/reference/androguard`, 2024. Accessed: 11/2024.

[19] 504ensicsLabs. Lime: Linux memory extractor (loadable kernel module for volatile memory acquisition). `https://github.com/504ensicsLabs/LiME`, 2012. GitHub repository.

[20] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining API-level features for robust malware detection in Android. In *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks (SecureComm)*, pages 86–103. Springer, 2013.

[21] AccessData Group LLC. *FTK Imager User Manual*, 2018.

[22] V. M. Afonso, P. L. de Geus, A. Bianchi, Y. Fratantonio, C. Krügel, G. Vigna, A. Doupé, and M. Polino. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *Network and Distributed System Security Symposium*, 2016.

[23] U. Aguirregomezcorta and J. A. García del Pozo. Mobile forensics for android devices: A systematic literature review (2015–2025). *Digital Forensics Research Review*, 2025.

[24] M. Ahammed. Clone detection to prevent software piracy in android play store. *GSC Advanced Research and Reviews*, 21:108–131, 12 2024.

[25] S. Alam, Z. Qu, R. D. Riley, Y. Chen, and V. Rastogi. Droidnative: Semantic-based detection of android native code malware. *CoRR*, abs/1602.04693, 2016.

[26] P. Albano, A. Castiglione, G. Cattaneo, and A. D. Santis. A novel anti-forensics technique for the android os. In *2011 International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA)*, pages 380–385, 2011. Early proposal of anti-forensic techniques and false-alibi construction on Android.

[27] M. Alecci, R. Cestaro, M. Conti, K. Kanishka, and E. Losiouk. Mascara: A novel attack leveraging android virtualization. *arXiv preprint arXiv:2010.10639*, 2020.

[28] A. Ali-Gombe, S. Sudhakaran, A. Case, and G. G. R. III. DroidScraper: A tool for android In-Memory object recovery and reconstruction. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 547–559, Chaoyang District, Beijing, sep 2019. USENIX Association.

[29] A. Ali-Gombe, S. Sudhakaran, R. Vijayakanthan, and G. G. Richard. crgb_mem: At the intersection of memory forensics and machine learning. *Forensic Science International: Digital Investigation*, 45:301564, 2023.

[30] 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 *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1347–1359, 2021.

[31] Andriller. *Andriller Forensic Toolkit*, 2023. `https://github.com/den4uk/andriller`.

[32] Android. Run Apps on the Android Emulator, 2024. https://developer.android.com/studio/run/emulator?hl=en.

[33] S. Aonzo, A. Merlo, Y. Fratantonio, and A. Ruggia. Obfuscapk: An open-source obfuscation framework for android apps. In *IEEE EuroS&P Workshops*, 2020. Used to study environmental deception and evasion.

[34] R. G. Arias et al. Systematic review: Anti-forensic computer techniques. *Applied Sciences (MDPI)*, 14(12), 2024. Recent systematic review organizing the anti-forensic literature and taxonomy.

[35] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Network and Distributed System Security Symposium (NDSS)*, 02 2014.

[36] S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song, and H. Yu. Samadroid: A novel 3-level hybrid malware detection model for android operating system. *IEEE Access*, 6:4321–4339, 2018.

[37] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 259–269, 2014.

[38] S. Banescu, C. Collberg, and A. Pretschner. Predicting the resilience of obfuscated code against symbolic-execution attacks via machine learning. In *26th USENIX Security Symposium (USENIX Security 2017)*, 2017. Evaluates obfuscation transformations vs. symbolic execution and proposes predictors of resilience.

[39] Belkasoft Research Team. Android forensics with belkasoft x: A practical guide, 2021.

[40] J. Bellizzi, A. Bruno, and F. Martinelli. Jit-mf: Just-in-time memory forensics for android devices. *Digital Investigation*, 42:301450, 2022.

[41] J. Bellizzi, E. Losiouk, M. Conti, C. Colombo, and M. Vella. Vedrando: A novel way to reveal stealthy attack steps on android through memory forensics. *Journal of Cybersecurity and Privacy*, 3(3):364–395, 2023.

[42] J. Bellizzi, M. Vella, C. Colombo, and J. Hernandez-Castro. Real-time triggering of android memory dumps for stealthy attack investigation. In M. Asplund and S. Nadjm-Tehrani, editors, *Secure IT Systems*, pages 20–36, Cham, 2021. Springer International Publishing.

[43] J. Bellizzi, M. Vella, C. Colombo, and J. Hernandez-Castro. Responding to targeted stealthy attacks on android using timely-captured memory dumps. *IEEE Access*, 10:35172–35218, 2022.

[44] R. K. Bhatia, S. Badhani, N. K. Sharma, A. K. Gupta, and N. Kumar. Xddroid: Exploring explainable ai for dynamic analysis of android malware. In *2025 12th International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 1–6, 2025.

[45] L. Borzacchiello, E. Coppa, D. Maiorca, A. Columbu, C. Demetrescu, and G. Giacinto. Reach me if you can: On native vulnerability reachability in android apps. In V. Atluri, R. Di Pietro, C. D. Jensen, and W. Meng, editors, *Computer Security – ESORICS 2022*, pages 701–722, Cham, 2022. Springer Nature Switzerland.

[46] L. Borzacchiello, M. Cornacchia, D. Maiorca, G. Giacinto, and E. Coppa. Droidreach++: Exploring the reachability of native code in android applications. *Computers Security*, 159:104657, 2025.

[47] C. D. Brant and T. Yavuz. A study on the testing of android security patches. In *2022 IEEE Conference on Communications and Network Security (CNS)*, pages 217–225, 2022.

[48] A. Brignoni and Y. K. Hofman. Aleapp: Android logs, events, and packages parser. `https://github.com/abrignoni/ALEAPP`, 2021. Open-source mobile forensic parser for Android artifacts.

[49] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, page 15–26, New York, NY, USA, 2011. Association for Computing Machinery.

[50] B. Carrier. *File System Forensic Analysis*. Addison-Wesley, 2005.

[51] B. Carrier. *Autopsy Digital Forensics Platform*, 2019. Open-source forensic suite, `https://www.sleuthkit.org/autopsy/`.

[52] A. Case, L. Marziale, and G. G. Richard. Memory forensics for android devices. In *Proceedings of the Digital Forensics Research Workshop (DFRWS USA)*, 2013.

[53] A. Case and G. G. Richard. Memory forensics: The path forward. In *Digital Investigation*, volume 20, pages 23–33, 2017.

[54] E. Casey. *Digital Evidence and Computer Crime: Forensic Science, Computers and the Internet*. Academic Press, Burlington, MA, 3rd edition, 2011.

[55] Cassavia, Nunziato and Caviglione, Luca and Guarascio, Massimo and Liguori, Angelica and Surace, Giuseppe and Zuppelli, Marco. Federated Learning for the Efficient Detection of Steganographic Threats Hidden in Image Icons. In *Pervasive Knowledge and Collective Intelligence on Web and Social Media*, pages 83–95. Springer Nature Switzerland, 2023.

[56] L. Caviglione, M. Gaggero, J.-F. Lalande, W. Mazurczyk, and M. Urbański. Seeing the unseen: revealing mobile malware hidden communications via energy consumption and artificial intelligence. *IEEE Transactions on Information Forensics and Security*, 11(4):799–810, 2015.

[57] L. Caviglione and W. Mazurczyk. Never mind the malware, here's the stegomalware. *IEEE Security & Privacy*, 20(5):101–106, 2022.

[58] L. Caviglione, S. Wendzel, and W. Mazurczyk. The Future of Digital Forensics: Challenges and The Road Ahead. *IEEE Security & Privacy*, 15(6):12–17, 2017.

[59] Cellebrite Ltd. *Cellebrite UFED User Guide and Product Overview*, 2023. Version 7.60, `https://www.cellebrite.com/en/ufed-ultimate/`.

[60] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang. IOTFUZZER: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *NDSS Symposium 2018*, 2018.

[61] K. Chen et al. Craxdroid: Automatic android app crash detection and analysis. In *IEEE 28th International Conference on Advanced Information Networking and Applications (AINA)*, 2014.

[62] Q. Chen et al. Deep learning in fuzzing: A comprehensive review. *IEEE Access*, 2022.

[63] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren. Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection. *IEEE Transactions on Information Forensics and Security*, 15:987 – 1001, 2020.

[64] Y. Chen, Z. Ding, and D. Wagner. Continuous learning for android malware detection. In *32nd USENIX Security Symposium (USENIX Security 2023)*, pages 1127–1144, 2023. Describes degradation of malware detector performance over time and proposes active / continual learning.

[65] H. Chung, J. Park, S. Lee, and C. Kang. Digital forensic investigation of cloud storage services. *Digital Investigation*, 9(2):81–95, Nov. 2012. Funding Information: This research was supported by Bio R&D program through the National Research Foundation of Korea funded by the Ministry of Education, Science and Technology (2011-0027732).

[66] M. Conti, S. Kumar, C. Lal, and S. Ruj. A survey on security and privacy issues of bitcoin. *IEEE Communications Surveys & Tutorials*, 20(4):3416–3452, 2018.

[67] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. 2012.

[68] J. Daoudi, T. F. Bissyandé, and J. Klein. Dexray: Simple, efficient and effective android malware detection via bytecode gray-scale imaging, 2021. arXiv preprint.

[69] D. Dell'Orco, G. Bernardinetti, G. Bianchi, A. Merlo, and A. Pellegrini. Would you mind hiding my malware? building malicious android apps with stegopack. *Pervasive and Mobile Computing*, 111:102060, 2025.

[70] A. Demontis, M. Melis, B. Biggio, D. Maiorca, D. Arp, K. Rieck, I. Corona, G. Giacinto, and F. Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Trans. Dependable Secur. Comput.*, page 711–724, jul 2019.

[71] A. Distefano, G. Merenda, and R. Santangelo. Anti-forensics for mobile devices: Exploring android-based techniques. *Digital Investigation*, 7:S25–S33, 2010. One of the early studies describing automated mobile anti-forensic techniques.

[72] A. Diwan and U. Sonkar. Visualizing the truth: a survey of multimedia forensic analysis. *Multimedia Tools and Applications*, 83(16):47979–48006, May 2024.

[73] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Communications of the ACM*, volume 57, pages 99–106, 2014.

[74] M. Farrokhmanesh and A. Hamzeh. A novel method for malware detection using audio signal processing techniques. In *2016 Artificial Intelligence and Robotics (IRANOPEN)*, pages 85–91. IEEE, 2016.

[75] P. Faruki, V. Laxmi, M. S. Gaur, and M. Conti. Android security: A survey of issues, malware penetration, and defenses. *IEEE Communications Surveys & Tutorials*, 17(2):998–1022, 2014. Includes discussion of native/JNI-based attacks.

[76] T. Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006. ROC Analysis in Pattern Recognition.

[77] R. Fedler, M. Kulicke, and J. Schütte. Native code execution control for attack mitigation on android. In *Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '13, page 15–20, New York, NY, USA, 2013. Association for Computing Machinery.

[78] H. Feng and K. G. Shin. Understanding and defending the binder attack surface in android. page 398–409, 2016.

[79] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. CodeBERT: A pre-trained model for programming and natural languages. In T. Cohn, Y. He, and Y. Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, Nov. 2020. Association for Computational Linguistics.

[80] H. Fereidooni, D. Demmler, A.-R. Sadeghi, et al. Droidbot: Automated and intelligent interaction for android apps. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1–7, 2017. Automated, stateful, randomized UI interaction for Android app testing.

[81] M. A. Ferrag, F. Alwahedi, A. Battah, B. Cherif, A. Mechri, N. Tihanyi, T. Bisztray, and M. Debbah. Generative ai in cybersecurity: A comprehensive review of llm applications and vulnerabilities. *Internet of Things and Cyber-Physical Systems*, 5:1–46, 2025.

[82] F. Franzen, T. Holl, M. Andreas, J. Kirsch, and J. Grossklags. Katana: Robust, automated, binary-only forensic analysis of linux memory snapshots. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '22, page 214–231, New York, NY, USA, 2022. Association for Computing Machinery.

[83] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 377–396, 2016.

[84] S. Freitas, R. Duggal, and D. H. P. Chau. Malnet: A large-scale image database of malicious software. In *CIKM '22: Proceedings of the 31st ACM International Conference on Information and Knowledge Management*, pages 4484–4493, 2022.

[85] Frida Project. Frida dynamic instrumentation toolkit, 2023. `https://frida.re`.

[86] Fridump Project. Fridump: Memory dumping tool for android, 2020. `https://github.com/Nightbringer21/fridump`.

[87] H. Fu and S. Wu. An android malware detection method based on native code and lstm. page 38–44, 2024.

[88] Genymobile. Genymotion android emulator, 2025. Version as of 2025; Android emulator with x86 images and hardware acceleration.

[89] GlobalWebIndex. Android device usage statistics 2025. `https://www.globalwebindex.com/reports/mobile-usage`, 2025. Accessed: 2025-10-23.

[90] P. R. C. GmbH. Excire foto 2025 (excire photo ai), 2025. AI-powered local photo management and content-based image retrieval software for Windows and macOS.

[91] P. Godefroid, H. Peleg, and R. Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[92] J. Gomez, N. Patel, and L. Verdoliva. Deepfake forensics: A survey on facial manipulation detection. *IEEE Access*, 11:64321–64356, 2023.

[93] Google. Ui/application exerciser monkey. `https://developer.android.com/studio/test/monkey`. Accessed: 2025-10-20.

[94] Google LLC. *Android Debug Bridge (ADB) Documentation*, 2023. `https://developer.android.com/studio/command-line/adb`.

[95] M. Gopinath and S. C. Sethuraman. A Comprehensive Survey on Deep Learning based Malware Detection Techniques. *Computer Science Review*, 47, 2023.

[96] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. DroidSafe: Android application information flow analysis via Systematic compositional analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[97] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. Adversarial perturbations against deep neural networks for malware classification. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 62–79, 2017.

[98] J. Gu, H. Zhu, Z. Han, X. Li, and J. Zhao. Gsedroid: Gnn-based android malware detection framework using lightweight semantic embedding. *Computers Security*, 140:103807, 2024.

[99] Guardsquare, 2023. Commercial Android application protection tool using multiple layers of obfuscation, encryption and runtime self-protection.

[100] Guardsquare. Proguard: Java class file shrinker, optimizer, obfuscator, and preverifier. https://www.guardsquare.com/en/products/proguard, 2024. Accessed: 2025-10-23.

[101] M. Guarnieri. *PCAPdroid: Android Network Capture Tool*, 2022.

[102] A. Heriyanto, C. Valli, and P. Hannay. Comparison of live response, linux memory extractor (lime) and mem tool for acquiring android's volatile memory in the malware incident. In *Proceedings of the European Conference on Cyber Warfare and Security (ECCWS)*, 11 2015.

[103] A. Hoog. *Android Forensics: Investigation, Analysis and Mobile Security for Google Android*. Syngress, 2011.

[104] H. Hu, Y. Huang, Q. Chen, T. Y. Zhuo, and C. Chen. A first look at on-device models in ios apps. In *Proceedings of the 22nd Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 123–136, 2023.

[105] Y. Huang, J. Lin, and J. Xu. Fassfuzzer: Fast automatic service security fuzzer for android. *Journal of Computers*, 33(2):197–210, 2022.

[106] Y. Huang, J. Wang, Z. Liu, Y. Wang, S. Wang, C. Chen, Y. Hu, and Q. Wang. Crashtranslator: Automatically reproducing mobile application crashes directly from stack trace. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[107] K. Ispoglou, D. Austin, V. Mohan, and M. Payer. FuzzGen: Automatic fuzzer generation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020)*, 2020.

[108] A. Joomye, M. H. Ling, and K.-L. A. Yau. A brief survey of deep learning methods for android malware detection. *International Journal of System Assurance Engineering and Management*, 2024.

[109] P. G. Kalauner. Analysis and bypass of android application anti-reverse engineering protections. Technical report, TU Wien (Technische Universität Wien), 2023. Practitioner study that inspects commercial anti-reverse tools (DexProtector, LIAPP, DashO) and their protections.

[110] Kaspersky Lab. How the necro trojan attacked millions of android users. https://www.kaspersky.com/blog/necro-infects-android-users/52201/, 2024. Threat analysis and timeline of the Necro Android trojan and its distribution via modified/repackaged apps.

[111] K. Kent, S. Chevalier, T. Grance, and H. Dang. Guide to integrating forensic techniques into incident response. Special Publication 800-86, National Institute of Standards and Technology (NIST), Gaithersburg, MD, 2006.

[112] G. C. Kessler. Anti-forensics and the digital investigator. 2007.

[113] D. S. Keyes, B. Li, G. Kaur, A. H. Lashkari, F. Gagnon, and F. Massicotte. Entroplyzer: Android malware classification and characterization using entropy analysis of dynamic characteristics. In *2021 Reconciling Data Analytics, Automation, Privacy, and Security: A Big Data Challenge (RDAAPS)*, pages 1–12, 2021.

[114] S. Khalid and F. B. Hussain. Volmemdroid—investigating android malware insights with volatile memory artifacts. *Expert Systems with Applications*, 253:124347, 2024.

[115] S. Khan, P. Krishnamoorthy, M. Goswami, F. M. Rakhimjonovna, S. A. Mohammed, and D. Menaga. Quantum computing and its implications for cybersecurity: A comprehensive review of emerging threats and defenses. *Nanotechnology Perceptions*, 2024.

[116] M. Kinkead, S. Millar, N. McLaughlin, and P. O'Kane. Towards explainable cnns for android malware detection. *Procedia Computer Science*, 184:959–965, 2021. The 12th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 4th International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops.

[117] T. Kröll, S. Kleber, F. Kargl, M. Hollick, and J. Classen. ARIstoteles – Dissecting Apple's Baseband Interface. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12972 LNCS, 2021.

[118] O. E. Kural, E. Kiliç, and C. Aksaç. Apk2audio4andmal: Audio based malware family detection framework. *IEEE Access*, 11:27527–27535, 2023.

[119] M. Kuštelega, R. Mekovec, and A. Shareef. Privacy and security challenges of the digital twin: systematic literature review. *JUCS - Journal of Universal Computer Science*, 30:1782–1806, 12 2024.

[120] J. Lessard and G. Kessler. Android forensic acquisition: A comparative study. In *Proceedings of the Conference on Digital Forensics, Security and Law*, 2010.

[121] J. Lessard and G. Kessler. Android forensic acquisition and analysis using custom recovery images. In *Proceedings of the Digital Forensics Research Workshop (DFRWS USA)*, 2011.

[122] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. volume 12, pages 1269–1284, 2017.

[123] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo. Libd: Scalable and precise third-party library detection in android markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 335–346, 2017.

[124] Y. Li, Y. Li, H. Cai, W. Huang, and X. Chen. Droidbot: A lightweight ui-guided test input generator for android. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 260–264, 2017.

[125] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. volume 19, pages 2244–2258, 2022.

[126] M. H. Ligh, A. Case, J. Levy, and A. Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory.* Wiley Publishing, 1st edition, 2014.

[127] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge. FANS: Fuzzing android native system services via automated interface analysis. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 307–323. USENIX Association, aug 2020.

[128] K. Luckow, M. Dimjašević, R. Grigore, C. Tinelli, and W. Visser. Jdart: A dynamic symbolic analysis framework. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, pages 442–445, 2016.

[129] W. Ma and B. Cui. Fuzzing IoT Devices via Android App Interfaces with Large Language Model. *Lecture Notes on Data Engineering and Communications Technologies*, 193, 2024.

[130] Z. Ma, H. Wang, Y. Guo, and X. Chen. Libradar: Fast and accurate detection of third-party libraries in android apps. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 653–656, 2016.

[131] Magnet Forensics Inc. *Magnet AXIOM: Comprehensive Digital Investigation Platform*, 2023. Product documentation, `https://www.magnetforensics.com/products/axiom/`.

[132] S. Mahdavifar, A. F. A. Kadir, R. Fatemi, D. Alhadidi, and A. A. Ghorbani. Dynamic android malware category classification using semi-supervised deep learning. *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, pages 515–522, 2020.

[133] V. Malhotra, K. Potika, and M. Stamp. A comparison of graph neural networks for malware classification. *Journal of Computer Virology and Hacking Techniques*, 20(1):53 – 69, 2024.

[134] P. Maniriho, A. N. Mahmood, and M. J. M. Chowdhury. Memaldet: A memory analysis-based malware detection framework using deep autoencoders and stacked ensemble under temporal evaluations. *Comput. Secur.*, 142(C), July 2024.

[135] E. Mariconti, L. Onwuzurike, P. Andriotis, E. D. Cristofaro, G. Stringhini, and G. Ross. MaMaDroid: Detecting Android malware by building markov chains of behavioral models. In *24th Network and Distributed System Security Symposium (NDSS)*, 2017.

[136] F. Mercaldo and A. Santone. Audio signal processing for android malware detection and family identification. *Journal of Computer Virology and Hacking Techniques*, 17(2):139–152, 2021.

[137] A. Merlo, A. Ruggia, L. Sciolla, and L. Verderame. You shall not repackage! demystifying anti-repackaging on Android. *Computers & Security*, 103:102181, 2021.

[138] Michael Hale Ligh and Andrew Case and Jamie Levy and AAron Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory.* Wiley, 2014.

[139] Microsoft Security Response Center. *AVML: Acquisition of Volatile Memory for Linux*, 2021. `https://github.com/microsoft/avml`.

[140] L. Minnei, H. Eddoubi, A. Sotgiu, M. Pintor, A. Demontis, and B. Biggio. Data drift in android malware detection. In *2024 International Conference on Machine Learning and Cybernetics (ICMLC)*, pages 157–163, 2024.

[141] L. Minnei, G. Piras, A. Sotgiu, M. Pintor, A. Demontis, D. Maiorca, and B. Biggio. An experimental analysis of semi-supervised learning for malware detection. 2025.

[142] I. A. Mohammad, A. O. Nasar, M. Alkhawaldeh, E.-U.-H. Qazi, and T. Zia. Anti-forensic challenges in digital forensics investigations: An overview of techniques and tools. 2024.

[143] T. Müller and M. Spreitzenbarth. Frost. In M. Jacobson, M. Locasto, P. Mohassel, and R. Safavi-Naini, editors, *Applied Cryptography and Network Security*, pages 373–388, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[144] A. Narayanan, L. Chen, and C. K. Chan. Addetect: Automated detection of android ad libraries using semantic analysis. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6, 2014.

[145] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, VizSec '11, New York, NY, USA, 2011. Association for Computing Machinery.

[146] A. Natarajan, M. Motani, B. de Silva, K. Yap, and K. C. Chua. Investigating network architectures for body sensor networks. In G. Whitcomb and P. Neece, editors, *Network Architectures*, pages 322–328, Dayton, OH, 2007. Keleuven Press.

[147] National Institute of Justice. Electronic crime scene investigation: A guide for first responders, second edition. Technical report, U.S. Department of Justice, Office of Justice Programs, 2008. https://www.ojp.gov/pdffiles1/nij/219941.pdf.

[148] Nightbringer21. fridump: A universal memory dumper using frida. `https://github.com/Nightbringer21/fridump`, 2019. GitHub repository, version 0.1.

[149] Z. Ning, G. Wang, Y. Zhou, X. Jiang, P. Ning, and X. Fu. Dexlego: Reassembleable bytecode extraction for aiding static analysis of android applications. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS) — Workshop / arXiv*, 2018. Extracts executed instructions and reassembles DEX to recover code hidden by packers / self-modifying behavior.

[150] D. B. Oh, S. Lim, S. Lee, Y. Jo, G. Choi, B. Kim, and H. K. Kim. Forensic analysis and evaluation of file-wiping applications in android os. *Journal of Forensic Sciences*, 2025. Empirical evaluation of file-wiping apps on Android; identifies traces and proposes an evaluation framework.

[151] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Trans. Priv. Secur.*, 22(2), apr 2019.

[152] E. Oriwoh, P. Sant, and G. Epiphaniou. Guidelines for internet of things deployment approaches — the thing commandments. In *IEEE International Conference on Cyber Security and Cloud Computing*, pages 1–6, 2013.

[153] Oxygen Forensics, Inc. *Oxygen Forensic Detective and Cloud Extractor*, 2022. Version 14.0, `https://www.oxygen-forensic.com/en/products/oxygen-forensic-detective`.

[154] F. Pagani and D. Balzarotti. Autoprofile: Towards automated profile generation for memory analysis. *ACM Trans. Priv. Secur.*, 25(1), Nov. 2021.

[155] B. Pala, L. Pisu, S. L. Sanna, D. Maiorca, and G. Giacinto. A targeted assessment of cross-site scripting detection tools. In *Proceedings of ITASEC 2023: The Italian Conference on CyberSecurity*, volume 3488 of *CEUR Workshop Proceedings*, pages xxx–xxx, Bari, Italy, 2023. CEUR-WS.org. Accessed: 2025-12-22.

[156] G. Palmer. A road map for digital forensic research. Technical report, Digital Forensic Research Workshop (DFRWS), August 2001. Technical Report DTR-T001-01 Final.

[157] Y. Pan, X. Ge, C. Fang, and Y. Fan. A Systematic Literature Review of Android Malware Detection Using Static Analysis. *IEEE Access*, 8:116363 – 116379, 2020.

[158] M. M. Pollitt. A history of digital forensics. In *Advances in Digital Forensics IV*, volume 285 of *IFIP International Federation for Information Processing*, pages 3–15. Springer, 2008.

[159] X. Qian, X. Zheng, Y. He, S. Yang, and L. Cavallaro. Lamd: Context-driven android malware detection and classification with llms, 2025.

[160] A. Rahali, A. H. Lashkari, G. Kaur, L. Taheri, F. Gagnon, and F. Massicotte. Didroid: Android malware classification and characterization using deep image learning. *Proceedings of the 2020 10th International Conference on Communication and Network Security*, 2020.

[161] N. Redini, A. Continella, D. Das, G. D. Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna. DIANE: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, May 2021.

[162] Z.-U. Rehman, S. N. Khan, K. Muhammad, J. W. Lee, Z. Lv, S. W. Baik, P. A. Shah, K. Awan, and I. Mehmood. Machine Learning-Assisted Signature and Heuristic-based Detection of Malwares in Android Devices. *Computers and Electrical Engineering*, 69:828 – 841, 2018.

[163] Rekall Project. *Rekall Memory Forensic Framework*, 2017. `https://www.rekall-forensic.com`.

[164] S. Rizvi, M. Scanlon, J. Mcgibney, and J. Sheppard. Application of artificial intelligence to network forensics: Survey, challenges and future directions. *IEEE Access*, 10:110362–110384, 2022.

[165] K. Ruan, J. Carthy, T. Kechadi, and M. Crosbie. Cloud forensics. In G. Peterson and S. Shenoi, editors, *Advances in Digital Forensics VII*, pages 35–46, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[166] A. Ruggia, E. Losiouk, L. Verderame, M. Conti, and A. Merlo. Repack me if you can: An anti-repackaging solution based on android virtualization. In *Proceedings of the 37th Annual Computer Security Applications Conference*, ACSAC '21, page 970–981, New York, NY, USA, 2021. Association for Computing Machinery.

[167] A. Ruggia, D. Nisi, S. Dambra, A. Merlo, D. Balzarotti, and S. Aonzo. Unmasking the veiled: A comprehensive analysis of android evasive malware. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '24, page 383–398, New York, NY, USA, 2024. Association for Computing Machinery.

[168] A. Ruggia, D. Nisi, S. Dambra, A. Merlo, D. Balzarotti, and S. Aonzo. Unmasking the veiled: A comprehensive analysis of android evasive malware. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '24, page 383–398, New York, NY, USA, 2024. Association for Computing Machinery.

[169] A. Ruggia, D. Nisi, S. Dambra, A. Merlo, D. Balzarotti, and S. Aonzo. Unmasking the veiled: A comprehensive analysis of android evasive malware. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '24, page 383–398, New York, NY, USA, 2024. Association for Computing Machinery.

[170] A. Ruggia, A. Possemato, S. Dambra, A. Merlo, S. Aonzo, and D. Balzarotti. The dark side of native code on android. *ACM Trans. Priv. Secur.*, 28(2), Feb. 2025.

[171] A. Sabbah, R. Jarrar, S. Zein, and D. Mohaisen. Empirical evaluation of concept drift in ml-based android malware detection. 2025.

[172] J. Samhi and A. Bartel. On the (in)effectiveness of static logic bomb detection for android apps. *IEEE Transactions on Dependable and Secure Computing*, 19(6):3822–3836, 2022.

[173] S. L. Sanna, C. Alcaraz, A. Sanna, G. Giacinto, and J. Lopez. Molife: Methodology, technologies, and challenges for mobile live intelligent forensics examination. *TechRxiv*, 2024.

[174] S. L. Sanna, C. Alcaraz, A. Sanna, G. Giacinto, and J. Lopez. Molife: Methodology, technologies, and challenges for mobile live intelligent forensics examination. *TechRxiv*, 2025.

[175] S. L. Sanna, A. Panzino, S. M. La Cava, S. Concas, L. Regano, D. Maiorca, G. L. Marcialis, and G. Giacinto. Pixel perfect or perfectly fake? exploring the robustness of digital forensic tools against facial deepfakes and morphed images. In *Proceedings of the 33rd European Signal Processing Conference (EUSIPCO 2025)*, pages 1357–1361, Isola delle Femmine, Palermo, Italy, 2025. EURASIP.

[176] S. L. Sanna, L. Regano, D. Maiorca, and G. Giacinto. Exploring the robustness of ai-driven tools in digital forensics: A preliminary study, 2024.

[177] S. L. Sanna, L. Regano, D. Maiorca, and G. Giacinto. Exploring the Robustness of AI-Driven Tools in Digital Forensics: A Preliminary Study, dec 2024. arXiv:2412.01363 [cs].

[178] S. L. Sanna, L. Regano, D. Maiorca, and G. Giacinto. Improving cybercrime detection and digital forensics investigations with artificial intelligence, 2025.

[179] S. L. Sanna, D. Soi, D. Maiorca, G. Fumera, and G. Giacinto. A risk estimation study of native code vulnerabilities in Android applications. *Journal of Cybersecurity*, 10(1):tyae015, 08 2024. In press.

[180] S. L. Sanna, D. Soi, D. Maiorca, and G. Giacinto. Are trees really green? a detection approach of iot malware attacks. *arXiv preprint arXiv:2506.07836*, 2025.

[181] Scapy Project. *Scapy Packet Manipulation Tool*, 2023. https://scapy.net/.

[182] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. volume 30, page 263–272, New York, NY, USA, Sept. 2005. Association for Computing Machinery.

[183] D. Sharma, G. Sharma, and A. Chattopadhyay. Ui-based dynamic behavior analysis for android malware detection. *Journal of Information Security and Applications*, 46:76–89, 2019.

[184] Skylot. Jadx: Dex to java decompiler, 2025. Version as of 2025; open-source tool for decompiling Android DEX and APK files to Java source.

[185] D. Soi, D. Maiorca, G. Giacinto, and H. Berger. Can you see me? on the visibility of nops against android malware detectors. arXiv preprint arXiv:2312.17356, 2023. https://arxiv.org/abs/2312.17356.

[186] D. Soi, A. Sanna, D. Maiorca, and G. Giacinto. Enhancing android malware detection explainability through function call graph apis. *Journal of Information Security and Applications*, 80:103691, 2024.

[187] D. Soi, A. Sanna, D. Maiorca, and G. Giacinto. Enhancing android malware detection explainability through function call graph apis. *Journal of Information Security and Applications*, 80:103691, 2024.

[188] D. Soi, S. L. Sanna, G. Benedetti, A. Liguori, L. Regano, L. Caviglione, and G. Giacinto. Analysis and detection of android stegomalware: the impact of the loading stage. In *Proceedings of the 2025 ACM Workshop on Information Hiding and Multimedia Security*, IH&MMSEC '25, page 35–45, New York, NY, USA, 2025. Association for Computing Machinery.

[189] D. Soi, S. L. Sanna, A. Liguori, M. Zuppelli, L. Regano, D. Maiorca, L. Caviglione, G. Manco, and G. Giacinto. On the Feasibility of Android Stegomalware: A Detection Study. *Italian Conference on CyberSecurity*, 2025. https://ceur-ws.org/Vol-3962/paper9.pdf.

[190] W. Song, H. Yin, C. Liu, and D. Song. Deepmem: Learning graph neural network models for fast and robust memory forensic analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 606–618. ACM, 2018.

[191] StatCounter. Mobile operating system market share worldwide. `https://gs.statcounter.com/os-market-share/mobile/worldwide`, 2025. Accessed: 2025-10-23.

[192] G. Suarez-Tangil, J. Tapiador, and P. Peris-Lopez. Stegomalware: Playing hide and seek with malicious components in smartphone apps. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8957:496–515, 2015.

[193] M. Sun and G. Tan. Nativeguard: Protecting android applications from third-party native libraries. In *Proceedings of the IEEE Cyber Security and Privacy Workshop (CSPW)*, WiSec '14, page 165–176, New York, NY, USA, 2014. Association for Computing Machinery.

[194] S.-T. Sun, A. Cuadros, and K. Beznosov. Android rooting: Methods, detection, and evasion. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '15, page 3–14, New York, NY, USA, 2015. Association for Computing Machinery.

[195] S.-T. Sun, A. Cuadros, and K. Beznosov. Android rooting: Methods, detection, and evasion. Association for Computing Machinery, 2015.

[196] J. Sylve, A. Case, L. Marziale, and G. G. Richard. Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, 8(3):175–184, 2012.

[197] J. Sylve, A. Case, L. Marziale, and G. G. Richard. Acquisition and analysis of volatile memory from android devices. *Digital Investigation*, 8(3):175–184, 2012.

[198] F. Taher, O. AlFandi, M. Al-kfairy, H. Al Hamadi, and S. Alrabaee. DroidDetectMW: A Hybrid Intelligent Model for Android Malware Detection. *Applied Sciences*, 13(13), 2023.

[199] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS '15)*, pages 1–15, 2015.

[200] P. Tarwireyi, R. Chisoro, and S. Musungwini. Malware detection using audio feature fusion and machine learning techniques. In *2023 International Conference on Intelligent Systems, Big Data and Infrastructure (ISBDI)*, pages 45–52. IEEE, 2023.

[201] tcpdump.org. *tcpdump Packet Analyzer*, 2023.

[202] The Volatility Foundation. Volatility framework — android memory forensics support. `https://github.com/volatilityfoundation/volatility/wiki/Android`, 2025. Official documentation page describing Android-specific memory forensics capabilities in the Volatility Framework.

[203] V. L. Thing, K.-Y. Ng, and E.-C. Chang. Live memory forensics of mobile phones. *Digital Investigation*, 7:S74–S82, 2010. The Proceedings of the Tenth Annual DFRWS Conference.

[204] Y.-J. Tung and I. G. Harris. A heuristic approach to detect opaque predicates that disrupt static disassembly. In *Workshop on Binary Analysis Research (NDSS BAR), co-located with NDSS*, 2020. Opaque-predicate detection; targets constructions that break static disassembly.

[205] C. . various authors. Audiostego: Audio file steganography tool. `https://github.com/danielcardeenas/AudioStego` (project page / CodeProject article), 2021. Open-source / community steganography tool for hiding text/files in MP3/WAV; discussed in tool surveys and steganography toolkits, https://github.com/danielcardeenas/AudioStego.

[206] A. S. N. C. Venkateswara Rao V. Survey on android forensic tools and methodologies. volume 154, pages 17–21, New York, USA, Nov 2016. Foundation of Computer Science (FCS), NY, USA.

[207] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 447–458, 2014.

[208] A. Walters. *The Volatility Framework: Advanced Memory Forensics*, 2012. `https://www.volatilityfoundation.org`.

[209] F. Wei, S. Roy, and X. Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[210] R. Wilson and H. Chi. A framework for validating aimed mobile digital forensics evidences. 2018.

[211] Wireshark Foundation. *Wireshark Network Protocol Analyzer*, 2023.

[212] R. Wiśniewski and C. Tumbleson. Apktool: A tool for reverse engineering android apk files, 2025. Version as of 2025; used for decompiling and rebuilding Android application packages.

[213] M. Wong and D. Lie. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. 2016.

[214] T. Wu, F. Breitinger, and S. O'Shaughnessy. Digital forensic tools: Recent advances and enhancing the status quo. *Forensic Science International: Digital Investigation*, 34:300999, 2020.

[215] X-Ways Software Technology AG. *X-Ways Forensics User Manual*, 2023. Version 21.6, `https://www.x-ways.net/forensics/`.

[216] N. Xi, Y. Zhang, P. Feng, S. Ma, J. Ma, Y. Shen, and Y. Yang. Gnndroid: Graph-learning based malware detection for android apps with native code. volume 22, page 1460–1476, Washington, DC, USA, Aug. 2024. IEEE Computer Society Press.

[217] C. Xia, M. Paltenghi, J. Tian, M. Pradel, and L. Zhang. Fuzz4ALL: Universal Fuzzing with Large Language Models. 2024.

[218] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. Chan. Ndroid: Toward tracking information flows across multiple android contexts. *IEEE Transactions on Information Forensics and Security*, 14(3):814–828, 2019.

[219] F. Yamaguchi, M. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *IEEE Symposium on Security and Privacy (SP)*, pages 797–812, 2014.

[220] L. Yan and H. Yin. Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic Android malware analysis. In *21st USENIX Security Symposium*, 2012.

[221] H. Yang, J. Zhuge, H. Liu, and W. Liu. A tool for volatile memory acquisition from android devices. *IFIP Advances in Information and Communication Technology*, 484:365 – 378, 2016. Cited by: 8.

[222] L. Yang, B. Zhang, and Y. Wang. Process-based memory extraction via frida for android forensics. In *Proceedings of the DFRWS EU*, 2019.

[223] S. J. Yang, J. H. Choi, K. B. Kim, and T. Chang. New acquisition method based on firmware update protocols for android smartphones. *Digital Investigation*, 14:S68–S76, 2015. The Proceedings of the Fifteenth Annual DFRWS Conference.

[224] H. Ye, S. Cheng, L. Zhang, and F. Jiang. Droidfuzzer: Fuzzing the android apps with intent-filter tag. In *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, MoMM '13, page 68–74, New York, NY, USA, 2013. Association for Computing Machinery.

[225] W. J. Youden. Index for rating diagnostic tests. *Cancer*, 3(1):32–35, 1950.

[226] M. l. Zalewski. memfetch: Utility to dump memory of a running process. `https://lcamtuf.coredump.cx/soft/memfetch.tgz`, 2007. Project page (memfetch.tgz) and FreeBSD port; accessed 2025-10-06.

[227] Zeek Project. *Zeek Network Security Monitor*, 2023.

[228] J. Zhang, E. Chengyuan, and A. Hu. A method of android application forensics based on heap memory analysis. In *Proceedings of the IEEE Cyber Physical Systems Conference*, 2018. Cited by: 0.

[229] Y. Zhang, Y. Yang, and X. Wang. A novel android malware detection approach based on convolutional neural network. In *Proceedings of the 2nd International Conference on Cryptography, Security and Privacy*, ICCSP 2018, page 144–149, New York, NY, USA, 2018. Association for Computing Machinery.

[230] Z. Zhao et al. Neufuzz: Efficient fuzzing with deep neural network. *IEEE Access*, 7:36340–36352, 2019.

[231] M. Zheng, P. P. C. Lee, and J. C. S. Lui. DroidAnalytics: A signature based analytic system to collect, extract, analyze and associate Android malware. In *Proceedings of the 2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 163–171, 2012.

[232] Q. Zhenxiao, Q. Yu, and Y. Heng. Logicmem: Automatic profile generation for binary-only memory forensics via logic inference. *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS'22)*, 2022.

[233] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang. Hybrid user-level sandboxing of third-party android apps. ASIA CCS '15, page 19–30, New York, NY, USA, 2015. Association for Computing Machinery.

[234] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting malicious apps in official and alternative android markets. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[235] N. Çarkacı. Binary-to-image: Convert executable binary files into rgb or grayscale images. `https://github.com/ncarkaci/binary-to-image`, 2025. GitHub repository, accessed 2025-10-06.

Acknowledgements

To everyone I met during my PhD, thanks for supporting me.

My PhD period has been supported economically by University of Roma Sapienza, University of Cagliari, Fondazione SERICS and Consorzio Interuniversitario Nazionale per l'Informatica (CINI).

Silvia Lucia Sanna, Rome, January 2026