Politecnico di Torino

Politecnico di Torino

Master of Science (M.Sc.) in Computer Engineering

A.a. 2024/2025

Graduation Session December 2025

Code Guardian

Fortifying Mobile Banking Applications

Supervisor
Prof. Giovanni Malnati

Company Supervisor
Salvatore Vitobello

Candidate
Angelo Squillino

Summary

Mobile banking applications have become a critical part of the modern banking ecosystem. Their handling of highly sensitive data and their ability to perform financial operations, coupled with their widespread adoption, make them very attractive targets for attackers. This thesis presents the design and development processes of Code Guardian, a static analysis tool aimed at supporting the vulnerability assessment of mobile banking applications distributed as `APK`(Android) and `IPA`(iOS) packages, focusing particularly on the evaluation of the obfuscation level. Code Guardian's analysis encompasses the inspection of the application's metadata, file system, embedded resources and binary executables and follows the guidelines of the **OWASP** Mobile Application Security(**MAS**) project. Firstly, a comprehensive evaluation was conducted in order to identify the most suitable components needed for the analysis workflow, ending up in the requirement of executing the analysis in a desktop environment to meet performance and compatibility constraints. Based on the results of this preliminary evaluation, the tool adopts a client-server architecture designed to combine the platform and computational requirements of the analysis with cross-platform accessibility for users. The server, developed using `Kotlin` and the `Ktor` framework, orchestrates the analysis processes by employing several techniques and external containerized tools such as `Ghidra`(in particular its headless version), `Semgrep` and a Large Language Model(LLM) in order to deeply dig into the package. On the other hand, the client is implemented using `Kotlin` and `Compose Multiplatform` allowing to share the codebase across multiple platforms (Android, iOS, desktop and web) while preserving high level performance thanks to its capability of compiling for different target platforms. To validate the effectiveness of Code Guardian, some tests were performed on a set of purposely vulnerable applications, including variants with and without obfuscation. The results proved that the tool is capable of successfully detecting the main vulnerabilities, giving recommendations to mitigate them and summarizing its findings in a structured security report.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Prof. Giovanni Malnati for his support and encouragement throughout the writing of this thesis. I am proud to have had the opportunity to work under the guidance of a person with the same infinite curiosity that I have, and that during his lectures has always stressed the importance of being a good man before being a good engineer. This is the teaching that I will carry with me the most.

I would also like to thank Salvatore Vitobello, my company supervisor, for his constant availability and precious pieces of advice that have been fundamental to successfully complete this work. I am grateful for having found in him a person with whom I can actively collaborate, share ideas and converse also beyond our professional sphere, rather than a simple supervisor. I am sure that I learned a lot from him and looking forward there is still much more to learn from him.

Table of Contents

List of Figures

Listings

Glossary

AAB
: Android App Bundle

adb
: Android Debug Bridge

AOT
: Ahead of Time

AP
: Application Processor

API
: Application Programming Interface

APK
: Android Package

ARC
: Automatic Reference Counting

ART
: Android Runtime

ASLR
: Address Space Layout Randomization

ATS
: App Transport Security

C2
: Command and Control

CA

Certificate Authority

Android CDD

Android Compatibility Definition Document

CLI

Command Line Interface

CMP

Compose Multiplatform

CPU

Central Processing Unit

CT

Certificate Transparency

DEP

Data Execution Prevention

DEX

Dalvik Executable

DNS

Domain Name System

DSL

Domain Specific Language

DVM

Dalvik Virtual Machine

ELF

Executable and Linkable Format

ENISA

European Union Agency for Cybersecurity

FBE

File Based Encryption

FDE

Full Disk Encryption

GUI

Graphical User Interface

GID

Group Identifier

HAL

Hardware Abstraction Layer

HTML

HyperText Markup Language

HTTP

HyperText Transfer Protocol

HTTPS

HyperText Transfer Protocol Secure

IBAN

International Bank Account Number

IPA

iOS App Store Package

IPC

Inter-Process Communication

JAR

Java ARchive

JIT

Just in Time

JSON

JavaScript Object Notation

KASLR

Kernel Address Space Layout Randomization

KMP

Kotlin Multiplatform

LLB

Low Level Bootloader

LLM

Large Language Model

NDJSON

Newline Delimited JSON

NDK

Native Development Kit

NIST

National Institute of Standards and Technology

NSA

National Security Agency

NX

No eXecute

OEM

Original Equipment Manufacturer

OHA

Open Handset Alliance

OS

Operating System

OWASP

Open Worldwide Application Security Project (Formerly Open Web Application Security Project)

MAS

OWASP Mobile Application Security

MASTG

OWASP Mobile Application Security Testing Guide

MASVS

OWASP Mobile Application Security Verification Standard

MASWE

OWASP Mobile Application Security Weakness Enumeration

PDF

Portable Document Format

PFS

Perfect Forward Secrecy

PIE

Position Independent Executable

REE

Rich Execution Environment

RELRO

Relocation Read-Only

ROM

Read Only Memory

SDK

Software Development Kit

SoC

System on a Chip

SQL

Structured Query Language

SRE

Software Reverse Engineering

SSE

Server Sent Events

SVG

Scalable Vector Graphics

TEE

Trusted Execution Environment

TLS

Transport Layer Security

TRNG

True Random Number Generator

UI

User Interface

UID

Unique Identifier

URL

Uniform Resource Locator

XML

eXtensible Markup Language

XN

eXecute Never

XNU

X is Not Unix

Chapter 1

Introduction

1.1 Context

Over the past decades, smartphones have become the center of gravity of communication, social interactions, commerce, advertising and also financial operations. In particular, mobile applications have completely transformed the way users interact with digital services providing seamless and easy access to almost every service. This shift led to an exponential increase in mobile banking applications usage, making them the primary interface for users' daily financial operations. As a direct consequence of this widespread adoption, the attack surface of mobile banking applications and their appeal as targets have increased.
According to the *European Union Agency for Cybersecurity*(**ENISA**) *Threat Landscape* 2024 report of the financial sector[1], 301 incidents suffered by European credit institutions and a sharp rise in mobile banking trojans were reported during the period going from January 2023 to June 2024. The report reveals a 200% year-over-year growth in malware families targeting mobile banking applications, increasing from 600 to 1800 applications affected globally and positioning the financial sector as the third most targeted. These malware keep evolving in their nature and attack strategy, becoming continuously more technically sophisticated and granting various possibilities to the attackers such as stealing credentials, performing fraudulent financial operations and even device takeover.
For example, during November 2023 the malware *Anatsa*[2] re-emerged, disguised as a cleaner application for Android. It initially gave the impression of being harmless to build trust in users and bypass detection and protections, but after a week an update that introduced malicious behaviour was released. The malware used to download a malicious `DEX` file from the Command and Control(C2) server of the attacker and load it dynamically in memory at runtime, allowing the attacker to gain full control of the infected device. Other examples of banking trojans are

ToxicPanda[3], *Brokewell*[4] and the newly discovered *RatOn*[5].
Overall, the natural appeal of the financial sector, the heterogeneous attack vectors and methodologies of the malware, and the constantly increasing number of incidents provide proof of the need to enhance the strategies to improve security in this domain.

1.2 Thesis Goal

The goal of this thesis is to design and develop a cross-platform, easy to use and automatic static analysis tool, named ***Code Guardian*** to assist the vulnerability assessment process of mobile banking applications for both Android and iOS applications. A key objective is to align the analysis methodology of the tool with the **OWASP** Mobile Application Security(**MAS**) project [6] guidelines to ensure a reliable baseline and cover the OWASP Top 10 Mobile Risks [7]. Additionally, the tool aims to also provide some insights about the overall quality of the application and above all to integrate the following layers of analysis:

- **Analyse application metadata**
  Inspect Android's `AndroidManifest.xml`, iOS's `Info.plist` and other files in order to detect misconfigurations or insecure settings that lead to security issues.
- **Analyse resources**
  Evaluate embedded string resources in order to detect any hardcoded secrets, credentials or sensitive data exposed in the package.
- **Analyse binaries**
  Assess the presence of binaries protection mechanisms, disassemble and decompile them, extracting strings, symbols and function bodies in order to evaluate obfuscation and potential sensitive data presence.
- **Analyse obfuscation**
  Evaluate the presence and effectiveness of obfuscation in the package binaries, assigning multiple scores based on the different obfuscation techniques employed.

1.3 Thesis Structure

The thesis is structured as follows:

- ***Chapter 2*** **- Mobile Platforms and Security Foundations**
  This chapter provides a comprehensive technical overview of the two mobile

platforms, describing their characteristics and security features. It also introduces some of the foundational principles applied by Code Guardian in its analysis.

- ***Chapter 3*** **- System Design and Technologies** This chapter illustrates the design process of the architecture of Code Guardian, explaining the rationale behind the tools and technologies chosen to implement it, and how these tools influenced the overall design. It also focuses on describing the individual components of the system and how they interact.
- ***Chapter 4*** **- Analysis Workflow** This chapter represents the core of the thesis, it describes in detail the analysis workflow of Code Guardian, explaining each of the performed steps and tasks and how they contribute and integrate among themselves to fulfill the thesis goal.
- ***Chapter 5*** **- Client Application** This chapter is dedicated to illustrating the client application of Code Guardian, focusing on its multiplatform nature, the technologies employed to implement it and how their limitations have been overcome. Furthermore, it shows the user interface and how it is capable of providing a great and consistent experience across all the supported platforms.
- ***Chapter 6*** **- Testing, Future Developments and Conclusion** This last chapter is focused on presenting some of the tests performed to validate the effectiveness of Code Guardian's analysis. It also discusses about potential future developments that could be implemented to furtherly enrich the tool and its capabilities.

The foundations about mobile platforms and security described in chapter 2, alongside the tests implemented are all based on the OWASP MAS[6], the Android official documentation[8] and the Apple Platform Security document[9].

Chapter 2

Mobile Platforms and Security Foundations

2.1 OWASP MAS

Alongside their Top 10 Mobile Risks, OWASP provides a comprehensive project revolved around mobile applications security called Mobile Application Security(MAS) project. The goal of this project is to define a standard for mobile applications called OWASP Mobile Application Security Verification Standard(**MASVS**), a list of common weaknesses of mobile applications referred as OWASP Mobile Application Security Weakness Enumeration(**MASWE**) and a testing guide named OWASP Mobile Application Security Testing Guide(**MASTG**) that covers a wide range of tests, techniques and best practices to assess the application's security.
In particular the MASTG includes detailed instructions to analyse, both statically and dynamically, the security of both Android and iOS applications. These instructions include tests categorized by different aspects of the application security such as data storage, cryptography, network communication, authentication and so forth. The guide is designed to cover both platforms, providing also suggestions about tools and benchmarking applications to support the analysis. The OWASP MAS is officially trusted by several institutions and companies like the *National Institute of Standards and Technologies*(**NIST**)[10] and Android itself. An overview of the OWASP MAS project is illustrated in Figure 2.1.

Figure 2.1: OWASP MAS Project

2.2 Android Platform Overview

Android is a Linux based open source platform used as a mobile operating system(OS). It is developed by the *Open Handset Alliance*(**OHA**) which is a consortium led by Google. Its openness made it the foundation of a wide ecosystem of devices built from various manufacturers, including mobile phones, tablets, wearables and so on.

2.2.1 Architecture

The bedrock of the software stack of Android, illustrated in Figure 2.2, is a Linux based kernel, extended to include some specific components like the `Binder`, which serves the Android Inter Process Communication(**IPC**) mechanisms.
Right upon the kernel, there is the Hardware Abstraction Layer(**HAL**) that provides standard interfaces for interacting with the device hardware. These interfaces are packed and exposed to the Java API Framework in shared libraries modules that the system loads as soon as they are requested.
What actually distinguishes Android is Android Runtime(**ART**), which is responsible for executing the `DEX` files containing Dalvik bytecode, which is an optimized version of Java bytecode. The Dalvik bytecode is designed to have a minimal memory footprint and it is derived from the Java bytecode by an additional compilation step. Starting from Android 5.0(API level 21), ART replaced its predecessor, the Dalvik Virtual Machine(**DVM**), introducing Ahead of Time(**AOT**) and Just in Time(**JIT**) compilation along with an optimized garbage collector. Beginning with

Android 7.0(API level 24), it introduced also a hybrid compilation approach called profile guided compilation which adapts compilation to the app's usage patterns, compiling AOT the most frequently used sections of the app.
Both the HAL and ART are built upon the Native C/C++ Libraries that the platform provides. They are typically accessible by the Java API Framework, which exposes the feature set of Android to system apps, or directly in native code through the Android Native Development Kit(**NDK**).

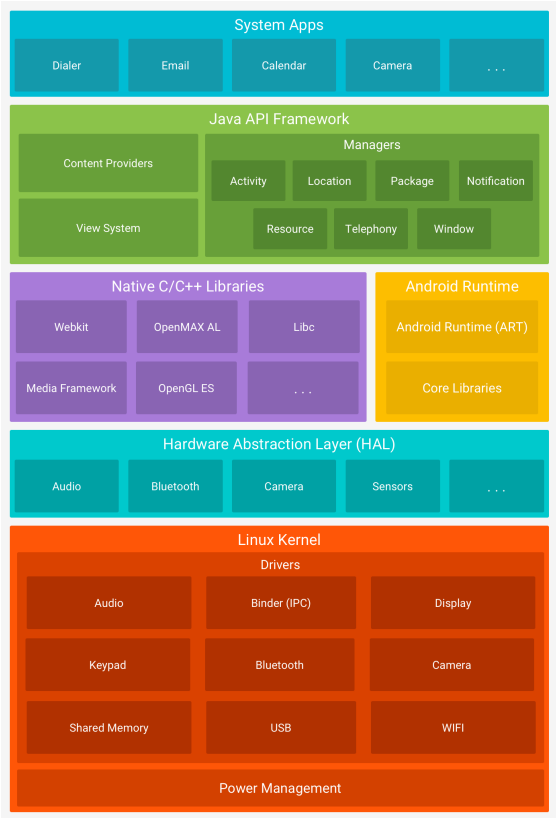

Figure 2.2: Android Architecture

2.2.2 APK structure

The Android Package(**APK**) is the main format used to distribute Android applications. However, Google introduced the Android App Bundle(**AAB**), a publishing format that can be uploaded to Google Play Store which then will be responsible for generating, signing and distributing different optimized APKs for each device architecture and configuration. Currently the Android system allows to install APKs also from different sources rather than the official store but Google announced that

starting from August 2026 this will not be possible anymore. This change aims to enhance the platform security and aligns the Android distribution policy with the iOS one.
An `APK` is basically an archive that encapsulates all the components required by an application, its main elements are listed below:

- **AndroidManifest.xml**
  The manifest is the key configuration file of the application. It is stored in the `APK` as a binary `XML` file containing the app's essential information like the package name, the components of the app along with their properties, the permissions that the app requires and declares, the network security configuration, deep links, backup behaviour and so on. The binary manifest that is within the `APK` is the result of compiling the original `AndroidManifest.xml` file and merging it with the manifests of the libraries that the app uses. It also includes some additional information injected by the build system such as if the app is debuggable or in test mode.

- **DEX files**
  The package includes one or more `DEX` files which contain the Dalvik bytecode obtained by compiling the source code of the application. They are named `classes.dex`, `classes2.dex` and so on.

- **Libraries**
  The `lib` directory includes the native libraries used by the application compiled for different architectures like `ARM`, `ARM64`, `x86` and so on. Each architecture has its own subdirectory.

- **Uncompiled Resources**
  The `res` directory contains the uncompiled resources of the application like images and layout files.

- **Assets**
  The `assets` directory contains raw asset files accessible by the application such as audio, video or fonts.

- **Compiled resources**
  The `resources.arsc` file contains the compiled resources and paths to resources not compiled in this file. It is a binary file that maps each resource ID to the actual resource data.

- **Metadata and signatures**
  The `META-INF` directory contains the manifest metadata, the signature, and the list of resources with their hashes which are used to verify the integrity of the package.

2.3 iOS Platform Overview

iOS is a mobile operating system developed by Apple that powers devices like iPhones and iPads. It is based on an open source operating system developed by Apple called Darwin. Its kernel is **XNU**(X is Not Unix), a kernel developed by Apple. A key difference from Android is that iOS benefits from a tight integration between hardware and software since these are both developed by Apple.

2.3.1 IPA structure

The iOS applications are distributed as `IPA` files within the App Store. An `IPA` file is essentially a `ZIP` archive containing all the necessary files for the application with a specific structure. The `IPA` archive includes the `Payload` folder which carries the actual application bundle with the `.app` extension. This bundle contains the main components required by the application, such as:

- **Executable**
  The main binary file of the application that contains the compiled code. It is a `Mach-O` file.
- **Information Property List**
  The `Info.plist` file is the main property list file that holds essential configuration information about the application, such as the app's bundle identifier, version number, supported devices and required permissions.
- **Frameworks**
  The `Frameworks` directory which includes all the frameworks the application depends on.
- **Resources**
  Resources comprehend images, strings files, sound files and any other custom resources. The bundle contains both non localized and localized resources, the first ones are stored directly in the top level directory of the bundle while the localized ones are organized in language specific subdirectories with the `.lproj` extension, one directory per each supported language.
- **Code signature**
  The `_CodeSignature` folder contains the code signature used by the system to verify the app's integrity.

Besides the `Payload` folder, the `IPA` package contains also other minor elements such as `iTunesMetadata` and `WatchKitSupport` directories which are not directly related to the application functionality.

2.4 Security

2.4.1 Bootstrapping

Since the very first moment a mobile device is powered on, the software integrity must be ensured to prevent the execution of malicious code that could compromise the whole device so the security strategy, of both Android and iOS, starts from the bootstrapping process.

Android

Verified Boot is the mechanism used by Android to ensure that all the executed software comes from a trusted source and has not been tampered with. It establishes a full chain of trust starting from the hardware protected root of trust up to the OS passing through the bootloader and the boot partition. The **Root of Trust** is the cryptographic key used to sign the Android distribution stored on the device. The Original Equipment Manufacturer(**OEM**) signs the Android version that will be distributed in the device with its private key which is known only by it while the public key is embedded in the device's hardware, in read only memory(**ROM**). Each component in the chain verifies the integrity of the following one before transferring the execution to it, if any verification fails, the device either will not boot or will boot in recovery mode. Verified Boot also supports rollback protection to prevent the installation of older Android versions.

iOS

The **Secure Boot Chain** of iOS is used to verify that the system and its components are written by Apple. It begins with the `Boot ROM`, which contains not modifiable code and the Apple Root Certificate Authority(**CA**) public key. The `Boot ROM` verifies the integrity of the Low Level Bootloader(**LLB**) which in turn verifies the `iBoot`. Then, `iBoot` is responsible for checking the signature of the iOS kernel. If any of these steps fails the device will not boot, additionally, in case of failure of the `Boot ROM` loading process, the device will enter a special recovery mode.

2.4.2 Package Signature

The **package signature** is used as a security measure to ensure the authenticity of the application package. It is checked by the system before installing, updating or executing an application to ensure that the package has not been tampered with.

Android

In Android, an `APK` can be signed with the following four different signature schemes:

- **v1**
  The v1 signature scheme is also known as `JAR` signature scheme since it is based on the same mechanism used to sign `JAR` files. It was the first signature scheme of Android and is supported by all of its versions.
  This scheme ignores some parts of the package like the `ZIP` metadata. Consequently, the verifier has to handle untrusted data before processing the signature and moreover it must decompress the whole `APK`, wasting time and memory.
  This signature scheme is insecure since the `JAR` can be modified without invalidating the signature and so it is not recommended to use.

- **v2**
  The v2 signature scheme covers the whole `APK` content resulting more secure than its predecessor. It has been introduced and supported starting from Android 7.0.

- **v3**
  The v3 signature scheme has been introduced in Android 9, bringing in support for key rotation and adding information about the supported `SDK` versions. It has been introduced to allow developers to update the signing key of the applications without obligating users to reinstall them. It consists of a singly linked list where each node signs the next one, so that newly introduced certificate is trusted because signed with the previous trusted one.

- **v4**
  The v4 signature scheme was introduced with Android 11 and ensures that all the devices with that version or higher have fs-verity enabled. This is a feature of the Linux kernel which allows to block reads of files of the `APK` that have been modified after its installation. This signature scheme is not a replacement of the previous ones, it requires at least a v2 or v3 signature to be present in the package.

An `APK` can be signed with multiple of these schemes to provide backward compatibility with older Android versions, for example devices running versions older than 7.0 will just ignore signature schemes different than the v1. The newer schemes are obviously preferred since they provide better security mechanisms.

iOS

In iOS, all the code that runs on an iPhone has to be signed by a valid Apple-signed certificate. It is mandatory for any application that is deployed on the App Store

but also for applications sideloaded outside of the official store, such as the ones installed during development. The difference between these two cases is that the first one needs a paid yearly subscription while the second can be done for free but still requires an Apple developer account. The code signing process involves three parts:

- **Seal**
  The seal is a group of hashes of multiple parts of the application's code. It can be used to detect alterations to the original code since any modification will consequently change the hash value.
- **Digital Signature**
  The digital signature is used to guarantee the integrity of the seal, it is created by encrypting the seal itself with the private key of the developer.
- **Requirements**
  The requirements are a set of rules that define the conditions that must be met in order to verify the code signature. They can be related to the verifier or directly specified by the signer.

2.4.3 Permissions Policy

Permissions are a security mechanism used by both platforms to protect sensitive resources such as camera and microphone from unauthorized access. In general, applications should request as few permissions as possible following the least privilege principle.

Android

In Android, each of the system defined permissions is associated with a **protection level** that indicates how sensitive the permission is and how the system handles it. The protection level defines if a permission is automatically granted at install time or it requires explicit approval from the user at runtime. Applications must declare the permissions they need in the manifest file but they can also define new custom permissions to be used by other applications when trying to interact with them using their components.

iOS

On the other hand, permissions cannot be assigned at install time in iOS, they are managed always asking at runtime for user consent when the application tries to access the protected resource for the first time. Applications must declare in the `Info.plist` file the permissions that they require, and provide a reason about why

they need them. Moreover this reason has become mandatory starting from iOS 10.

2.4.4 Hardware-based isolated environments

Sensitive tasks like cryptographic operations or random number generation are performed in **hardware-based isolated environments** to protect them from potential threats coming from the main operating system.

Android

Android has to deal with a lot of different OEMs and devices so it adopts different approaches to provide isolated environment. The Android Compatibility Definition Document(**CDD**), which defines the standard requirements that each Android device has to meet in order to be considered compatible, specifies that the devices have to provide at least a Trusted Execution Environment(**TEE**).
A TEE is a secure area of the main application processor(**AP**) on which an isolated OS runs alongside the main one, which instead executes in Rich Execution Environment(**REE**). The isolation between these two environments is enforced with hardware mechanisms provided by the processor, for example ARM processors implement the ARM TrustZone technology to achieve this strong separation. This approach is virtual since both the REE and TEE share the same CPU and memory even if the REE, which is considered untrusted, is prevented from accessing the resources of the TEE by hardware. Google also provides an open source implementation of a TEE OS called **Trusty** which can be directly used or as a baseline to build other custom TEE operating systems.
The CDD also establishes that devices can and should support an additional stronger isolated environment called **StrongBox**, which is a completely separated chip embedded in the device. StrongBox includes its own CPU, storage and True Random Number Generator(TRNG) setting precise boundaries to cryptographic sensitive operations. It is designed to be tamper resistant and autonomous so it is intrinsically more secure than a TEE given that an eventually compromised REE would be running on a completely different hardware. However, currently StrongBox is not mandatory but as stated in the CDD it is highly recommended and it will likely become a mandatory requirement in future releases.

iOS

In contrast to Android, Apple has full control over both hardware and software of iPhones so it can adopt a unified approach to provide secure isolated environment. In fact, all the recently released iOS devices include the **Secure Enclave**, which is a dedicated subsystem integrated into the main System on a Chip(**SoC**) which

provides hardware isolation from the main processor. Secure Enclave has its own Boot ROM, its own microkernel based OS called **sepOS** and its own TRNG. It does not have an actually separated memory but it uses a dedicated region of the main memory which is protected by the Memory Protection Engine that prevents the main processor from accessing it. The iOS devices also include two AES 256 bits cryptographic engines to perform the cryptographic operations efficiently, one of them is exclusively dedicated to Secure Enclave while the other is used by the main processor.

2.4.5 Cryptography

Cryptography is fundamental to ensure data confidentiality both at rest and in transit. Android and iOS have adopted similar approaches that are described in the following sections.

Android

Starting from Android 6.0, Google forced devices to support storage encryption with some exceptions for low end devices.
Android 5.0 introduced Full Disk Encryption(**FDE**) which uses a single key to perform cryptographic operations on the user data partition. This key is protected by the device lockscreen credentials set by the user. However this paradigm has important limitations since a just rebooted device could not receive calls, messages or alarms until the user enters the credentials to decrypt the storage.
Hence, Android 7.0 brought in File Based Encryption(**FBE**) which allows different files to be encrypted and decrypted independently, since these operations are performed with different keys. This type of encryption comes along with **Direct Boot** support that allows to overcome the limitations of FDE giving access to alarms, calls or accessibility services even if the device has not been unlocked by the user.

iOS

Apple devices support encryption since the introduction of the iPhone3GS. All iOS devices are equipped with two AES 256 bits keys: the first one is the device's unique identifier(**UID**) which is fused in the SoC during the device production and it is completely unique for each device, the second one is the device group identifier(**GID**) which is common to all the devices sharing the same SoC model. These keys are not accessible by software in any way, the UID is not even known to Apple because it is not recorded during the manufacturing process. The GID is used to prevent tampering with the firmware and to perform some cryptographic operations that are not strictly related with the user's data, while the UID is part

of the key hierarchy used to protect the file system, so that even mounting the storage on a different device would not allow to access the data since the UID would be different.
The file based encryption used by iOS is called **Data Protection**, it gets enabled as soon as the user sets a passcode on the device. Data Protection is based on a key hierarchy where each file is associated with a unique per-file key which is used to encrypt its contents. Each file is also associated with a protection class that specifies when the file should be accessible, based on the device lock state. For example, some files like those related to alarms can be accessed even if the device is locked while others become accessible only once the device has been unlocked. The per-file key is wrapped with the class key corresponding to the protection class of the file and saved in the file's metadata which then are encrypted with the file system key. The UID and the user's passcode are used to derive the keys required to unlock the class keys that are stored in the keybag. For example, the files assigned to the protection class that require the device to be unlocked in order to access them, will need both UID and password to derive the key to unlock that class key, therefore if the file is associated with the class that does not require the device to be unlocked, the file is still encrypted but its class key needs only the UID to be accessible, the passcode is not required since the file can be accessed even if the device is locked. All of these operations are obviously performed inside the Secure Enclave to ensure a bulletproof environment.

2.4.6 Network Communications

Almost every modern mobile application relies on some network interaction to implement its functionalities. Often, these communications involve sensitive data exchanges like for example any authentication process with a server or payments, therefore they need to be properly managed and secured by the operating system and the application itself.

Android

Starting with Android 9, every network channel is established using **TLS** by default. Applications targeting this or newer versions can still decide to use clear text traffic by explicitly declaring it in their manifest or network security configuration file but it is strongly discouraged. Moreover, Android 10 furtherly tightened this policy by enabling the 1.3 version of TLS by default. Alongside the introduction of this TLS by default policy in Android 9, it was introduced also system wide **DNS over TLS** support which allows to perform DNS lookup over a secure channel established using TLS so that no sensitive information can be exposed.

iOS

Apple introduced App Transport Security(**ATS**) starting with iOS 9 as a collection of security policies that aim to prevent insecure network connections. ATS enforces a set of minimum requirements to secure the connections such as the usage of TLS 1.2 or higher, data encryption with AES-128 or 256 and support of Perfect Forward Secrecy(**PFS**). These restrictions can also be disabled by applications configuring global or specific exceptions in their `Info.plist` file to lighten some of these requirements, like lowering the minimum TLS version for instance. Nevertheless, starting from 2017, the Apple App Store requires to provide justifications for these exceptions when the package is submitted to the store. It is preferred to avoid disabling these security restrictions and instead properly configure the server to comply with the requirements imposed by ATS.

2.4.7 Sandboxing

Sandboxing is a security mechanism extensively used in both Android and iOS to isolate applications and restrict them from accessing resources or data belonging to other applications or the system itself. In this way every application runs in its own isolated environment and cannot interfere with the other ones.

Android

Android's sandboxing model has its foundations in the Linux multi user separation. Each Android application is assigned to a separated Linux user with a unique user ID, in this way the applications are naturally isolated from each other and from the system, leveraging the enforcements made by the Linux kernel. At installation time, a new directory named after the app's package name is created under the `/data/data/` path and it is made accessible for read and write operations only by the user corresponding to that application. Actually, there is the possibility for applications signed by the same certificate to share a single common sandbox by explicitly declaring this intention in their manifests. In this case the two applications will share the same user ID becoming able to mutually access the data of each other.
Android relies also on the groups system of the Linux kernel by mapping each permission to one or more Linux groups that are assigned to the applications based on their granted permissions, therefore ensuring that only applications having the required permissions can access the corresponding system resources.

iOS

The iOS app Sandbox, also called **Seatbelt**, is also enforced at kernel level but it is not based on the multi user separation like in Android. Only a few applications of the system run as root user and the third party applications run under an unprivileged user called `mobile`. The applications are constrained in a dedicated container and they are able to access only their own directory, which is randomly assigned when the app gets installed, and a limited set of the system APIs.

2.4.8 Inter process Communication

IPC is the mechanism that allows processes to communicate with each other. Since in mobile operating systems the applications are all sandboxed and isolated from each other, there is no natural way for them to securely interact without using specific IPC channels and mechanisms provided by the operating system. The cooperation obtained by IPC is an incredible source of new features, it integrates the capabilities of multiple applications but the exposure required in order to communicate across the sandbox has to be carefully managed and restricted to avoid malicious exploitation.
Android provides IPC communication through the `Binder` framework which differs from the traditional Linux IPC and has been built specifically for Android. Exploiting the `Binder`, applications have the possibility to request services to other applications as if they were performing a normal procedure call. A higher level IPC mechanism built on top of the `Binder` is the `Intent`, which can be used by applications to asynchronously communicate, requesting services offered by components of other applications which in turn are exposed via specific Intent filters declared in the manifest. This exposure of the components is regulated by the permissions that allow to restrict the access to the services exposed by a component via the `Intent` mechanism only to the applications owning the required permissions.
In iOS, IPC is not as comprehensive as Android's Binder, it is limited to very few mechanisms contributing to supervise the attack surface better since the applications components are not exposed as much as in Android.

2.4.9 Binary Protections

Several attacking techniques like the buffer overflow can be exploited by attackers to tamper with the normal execution flow of an application and gain control of it. In general these attacks are counterable directly implementing some **binary protections** that both Android and iOS provide and are explained below.

- **Address Space Layout Randomization(ASLR)**

ASLR is a mechanism that randomizes the memory location of a program's code, heap and stack in order to make it nearly impossible for an attacker to know the exact memory addresses where the program segments are located. This technique is available on Android since version 4.1 and it has been extended to cover also the kernel(KASLR) in Android 8.0. This protection is feasible only if the binary is a Position Independent Executable(**PIE**), which is a binary that can be loaded to any memory address, so a binary for which the address binding is not done at compile time. Additionally, since Android 5.0, all the applications must be compiled as PIEs and similarly Xcode automatically compile applications with the ASLR enabled by default.

- **Data Execution Prevention(DEP)**
  Attacks such as buffer overflows often aim to inject and execute malicious code in the memory regions of the application. A legitimate program would never need to execute code from the stack or heap memory regions so marking them as non executable prevents these exploitations without affecting the normal behaviour of the application. This protection is called Data Execution Prevention(DEP) and it is implemented using the NX(No eXecute) or XN(eXecute Never) bits to mark segments as non executable. This technique requires hardware support and even if the names associated with this protection are different among architectures, the underlying principle is the same.

- **Stack Canaries**
  Stack canaries are special random values placed in the stack between local variables and the return address of a function. They are unique among all the functions during a single program execution and their purpose is to detect any modification to the regular stack. When a function is called, a canary value is pushed into the stack right before the return address and before the function returns, the canary is checked to see if it has been altered with respect to the original one. If it has, the canary has been killed and so the program immediately terminates throwing a segmentation fault, preventing so the execution of potentially tampered code.

2.4.10 Obfuscation

Obfuscation is a software protection process that aims to reduce the software intelligibility in order to make it difficult to reverse engineer it for an attacker. Obfuscation is not an absolute lock which ensures that an attacker will never be able to reverse engineer the application, it just makes the process considerably harder and discouraging to the point that it is not worth the effort. There are several obfuscation techniques, some of the most common ones which are taken into consideration by Code Guardian are described below:

Symbols Obfuscation

Normally, the compiler produces the binary symbols starting from class names, function names and variables. If no obfuscation is applied, these names correspond to the ones defined in the source code and so they give hints about the purpose of the function or variable. In order to hide this semantic information the symbols are usually substituted with some meaningless characters like a single character or random strings as shown in the Listing 2.1.
The debug symbols are special symbols inserted by compilers to ease the development process, in particular its debugging. They must not be present in production applications since their presence can reveal implementation details, ease the disassemblers and decompilers job and consequently the reverse engineering process, therefore obfuscators should aim to strip them out of the binaries.

SymbolsObfuscation.java

```java
//Regular
public boolean executeWireTransfer(String beneficiaryIban, double amount, String currency) {
    if (UserAccount.hasSufficientFunds(amount)) {
        BankAPI.sendPayment(beneficiaryIban, amount, currency);
        return true;
    }
    return false;
}


//Obfuscated
public boolean a(String s1, double d1, String s2) {
    if (p.k(d1)) {
        n.q(s1, d1, s2);
        return true;
    }
    return false;
}
```

Listing 2.1: Symbols Obfuscation Example

Strings Obfuscation

By default, strings within the source code are stored in plain text in the application's binary and this behaviour can be dangerous because they may contain hardcoded secrets such as API keys. To mitigate this aspect, it is possible to encode or encrypt them and compute the real string at runtime before using it as shown below in the Listing 2.2. Note that encoding is just an illusory obfuscation since it is easily reversible so it is never used in practice.

StringsObfuscation.java

```java
//Regular
public void connectToBank() {
    String endpoint = "https://api.secure-bank.com/v1/transfer";
    System.out.println("Connecting to: " + endpoint);
}


//Obfuscated
public void connectToBank() {
    String endpoint = Decryptor.decode("x8s7f9s87df9s87df9s87d");
    System.out.println("Connecting to: " + endpoint);
}
```

Listing 2.2: Strings Obfuscation Example

Dead Code Injection

Dead code injection does not modify the original program behaviour but introduces pieces of code that will never be executed, it is often coupled with opaque predicates introduction in the code. Its objective is to bring in useless code to increase the reversing complexity.

Control Flow Manipulation

Control flow of a function can be manipulated to make it significantly harder to understand. It is possible to introduce opaque predicates, to apply control flow flattening or other modifications in order to introduce complexity in the function body. These manipulations transform the program's flow making it really challenging to understand statically. A typical consequence of the control flow flattening and opaque predicates introduction is increased cyclomatic complexity. A complete example including control flow manipulation and dead code injection (as well as symbols obfuscation) is shown in the following Listing 2.3.

FunctionsObfuscation.java

```java
//Regular
public boolean executeWireTransfer(double amount) {
    if (balance >= amount) {
        balance = balance - amount;
        logTransaction();
        return true;
    } else {
        showError();
        return false;
    }
}


//Obfuscated
public boolean a(double d1) {
    int flowCtrl = 12;
    boolean retVal = false;
    double fakeTax = 0.0;
    while (true) {
        switch (flowCtrl) {
            case 12:
                // Dead Code Injection
                fakeTax = d1 * 0.22;
                if (fakeTax < 0) { flowCtrl = 99; } // Opaque, never taken
                if (this.b >= d1) {
                    flowCtrl = 5;
                } else {
                    flowCtrl = 8;
                }
                break;
            case 5:
                this.b = this.b - d1;
                //Dead Code
                long dummyTime = System.currentTimeMillis();
                if (dummyTime % 2 == 0) { fakeTax += 1; }
                this.z();    //logTransaction() renamed in z()
                retVal = true;
                flowCtrl = 30;
                break;
            case 8:
                this.e();    //showError() renamed in e()
                retVal = false;
                flowCtrl = 30;
                break;
            case 30:
                return retVal;
        }
    }
}
```

Listing 2.3: Functions Obfuscation Example

Chapter 3

System Design and Technologies

3.1 Design Goals and Tools Selection

Code Guardian's ultimate goal is to provide a multiplatform tool capable of automatically analyse in depth Android and iOS applications in order to provide some information about their quality and especially about their security adhering to the guidelines outlined by OWASP MASTG.
In this scenario, the analysis process has to deal with the wide range of file types that are present in mobile applications packages including binary `XML`, property lists, resources, `DEX` files, `Mach-O` binaries and so on. As a result, Code Guardian must be able to perform multiple specialized tasks on different file types such as parsing information from binary `XML` and property lists files and disassemble binaries of different types.

3.1.1 Tools and Technologies Selection

Since the first requirement of Code Guardian is to be cross-platform, the chosen technology for its development is `Kotlin Multiplatform`(`KMP`), which allows to share the codebase across mobile platforms, desktop and web.
The tasks of the analysis process require some specialized tools and libraries to be successfully accomplished. For this reason, the first step that took place in the design of Code Guardian was benchmarking and evaluation of available options, suggested also by OWASP MASTG, that could be integrated in the project to fulfill the various required functionalities.

Packages Extraction

First of all, the packages extraction required different approaches for `APK` and `IPA` files. The `IPA` files are just `ZIP` archives with a different extension, so the extraction task could be performed with any `ZIP` manipulation library. Inversely, `APK` files needed a specialized tool since they contain compiled resources and are not simple `ZIP` archives. For this reason, several tools were evaluated, considering factors such as provided features and ease of integration. For example, `Androguard` was considered as a possible tool for `APK` extraction given its additional capabilities, but then was subsequently discarded since it is a pure `Python` library and so its integration in a `Kotlin Multiplatform` project would have been not straightforward.
Therefore, for `APK` extraction it was adopted **`apktool`**[11], an open source tool capable of extracting and decoding `APK` packages. This tool is written in `Java` and distributed as a Java Archive(**JAR**) file resulting easy to integrate in a `Kotlin` project. Some additional minor tool has also been selected for specific tasks such as `vd2svg` for converting Android vector drawables to SVG in order to be able to extract the app's icon when it is an Android vector drawable.

Manifest and Info.plist Parsing

The information parsing from Android's `AndroidManifest.xml` and iOS's `Info.plist` files required the capability of reading binary `XML` and `plist` files. In this case, the choice has been relatively straightforward converging on **`jdom`** for `XML` and **`ddplist`** for `plist` files. Both of them are `Java` libraries, easy to integrate in `Kotlin` and capable of parsing information from both plain text and binary versions of the files.

Resources Analysis

String resources as well are included as `XML` files in Android applications and as property lists in iOS ones. Therefore, the same libraries used for the parsing step are suitable also for this task. Additionally, for the secrets detection in string resources there were evaluated `GitLeaks`[12] and `TruffleHog`, both open source tools recommended by OWASP MASTG. They are really similar but they focus on source code scanning and so they are not perfectly suitable to simple string resources. Consequently, it has been preferred to adopt a completely custom implementation.

Binaries Analysis

The binaries analysis step is the most complex one since it involves multiple tasks such as disassembling, strings and symbols extraction, control flow evaluation

and so on. OWASP MASTG suggests several tools for this purposes, the main ones are **Radare2**[13] and **Ghidra**[14]. Both of them are mature software reverse engineering(**SRE**) frameworks suitable to handle the binary types that are present in the mobile packages but they have some differences which influenced the final integration decision.
Radare2 is a lightweight command line interface(**CLI**) tool whose main advantage is the ease of scripting via one of its components called `r2pipe`. Nevertheless, it lacks an important feature, the decompiler.
Ghidra instead, which is developed by the *United States of America's National Security Agency*(**NSA**), besides the disassembler includes a powerful decompiler and is scriptable as well thanks to its headless version. For this specific reason, Ghidra has been preferred over Radare2 as a starting point for the binaries analysis. Actually there is the possibility with some plugins to integrate Ghidra in Radare2 and viceversa but this would have added unnecessary complexity. On the Android side there was also the possibility to rely on `apktool`'s built-in disassembler `baksmali` but this option has been discarded in favor of a unified and more powerful approach for both Android and iOS platforms using `Ghidra`, also because `baksmali` cannot even disassemble ELF files, it is dedicated only to DEX files.
The decompiled code could then be analysed to detect security weaknesses such as the use of poor cryptographic functions and to meet with this goal, **Semgrep**[15] has been adopted. A key feature that influenced this choice is that `Semgrep` allows to define custom rules to tailor the analysis to specific needs.

Obfuscation Evaluation

The obfuscation evaluation task makes use of all the data extracted in the binaries analysis step. The evaluation considers several obfuscation techniques including symbols and strings obfuscation. These techniques are designed to remove meaning from symbols and strings within the binaries. Even though features such as entropy and presence of special characters can be easily evaluated with custom logic, assessing the semantical meaninglessness of symbols and strings requires a different approach. To evaluate this aspect, Code Guardian integrates with a Large Language Model(**LLM**) which is queried with the symbols and strings to classify them as meaningful or meaningless. The result of this classification is a key factor in the obfuscation evaluation of each symbol or string since their meaning is what obfuscation aims to conceal.

3.2 Architecture Overview

Code Guardian was originally meant to be a monolithic application, completely developed with `Kotlin Multiplatform` for the purpose of being able to execute it

on desktop, web and mobile devices as a single unit.
However, the just described tools that the analysis has to integrate to accomplish its goals require a lot of computational resources and above all, some of them are not even compatible with all the platforms that Code Guardian targets. To address this need, the final architecture of Code Guardian is a client-server one, where the client is designed as a lightweight cross-platform application developed with `Kotlin Multiplatform` and the server is a more powerful backend dedicated to perform the actual analysis tasks. The server is implemented using `Kotlin` with the `Ktor` framework, a choice driven by consistency with the client's technology stack and the possibility of sharing some code between client and server such as data models, thereby overcoming the need of additional data transformations on either side. A visual representation of the high level architecture is shown in Figure 3.1.

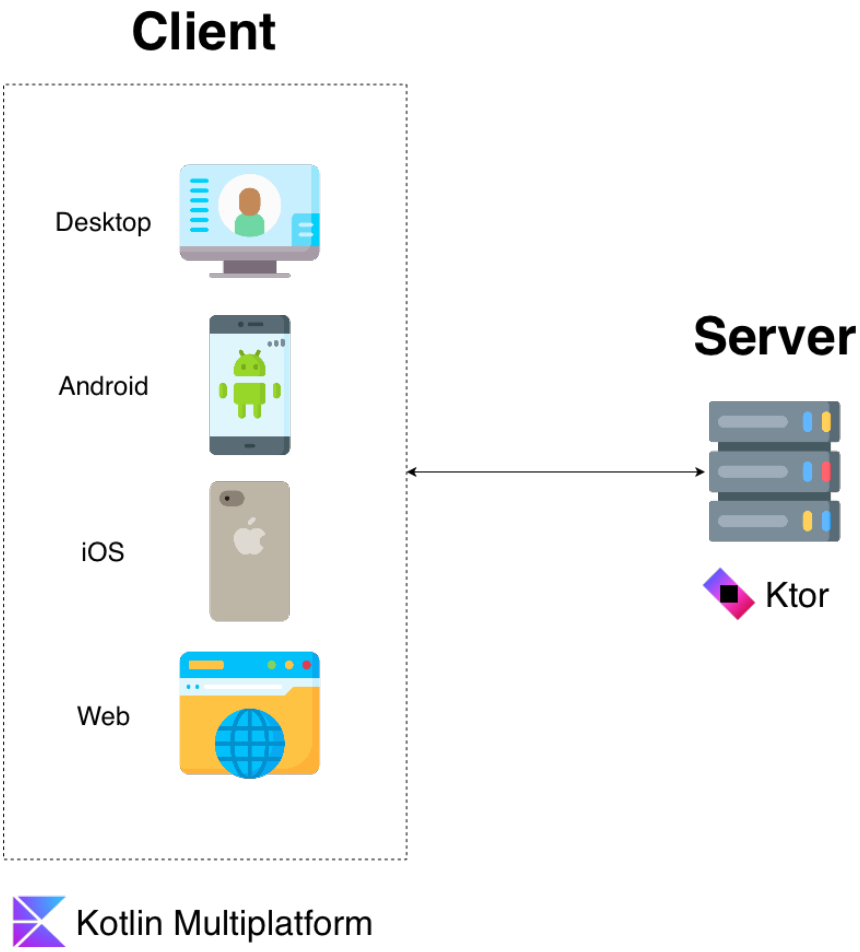

Figure 3.1: High Level Architecture

3.3 Server Architecture

The server is the brain of Code Guardian since it is responsible for orchestrating the analyses process. The persistence of analyses data is managed through a `PostgreSQL` database and `Exposed`. The latter is a `Kotlin` SQL library used in its Domain Specific Language(**DSL**) flavor that allows to define the database schema, perform queries and transactions without writing raw SQL statements, benefiting so from `Kotlin`'s type safety.
The analysis process strictly requires the tools selected and described in the previous section, some of them are directly employed as libraries or `JAR` files while others such as headless `Ghidra` and `Semgrep` are containerized with `Docker` and invoked by the server as needed through `Docker` CLI commands. The LLM is accessed through its API and it is provided by Azure AI Studio.
The described architecture of the server is visually shown in the following Figure 3.2.

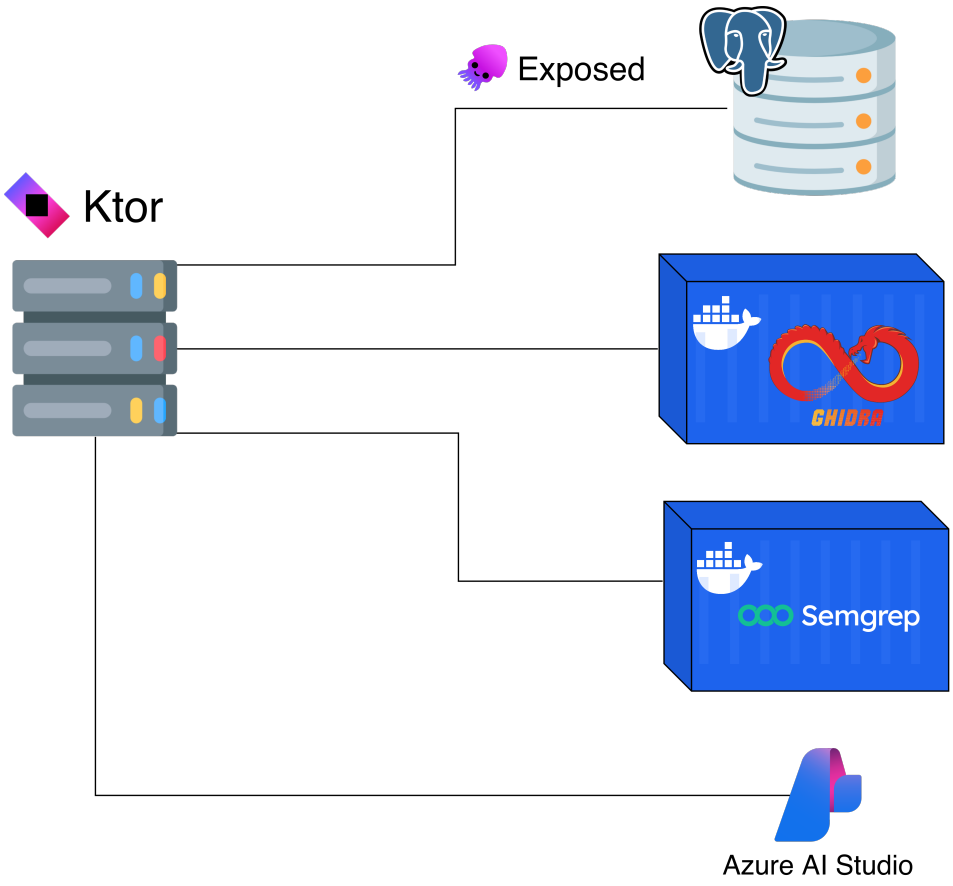

Figure 3.2: Server Architecture

3.4 Client-Server Communication

Almost all the interactions between clients and server are standard **HTTP** request-response exchanges,with a single exception represented by the real time analysis progress updates that are sent from the server to the clients using Server Sent Events(**SSE**).
`Ktor` natively supports SSE on both client and server side leveraging the `Kotlin`'s `Flow`. Clients interested in receiving updates regarding a certain analysis can subscribe to that analysis with a SSE endpoint exposed by the server. The server initially will send the current status of the requested analysis as first event, then if the analysis is still ongoing it will keep the connection open and as soon as a new progress update is available it will send it to the clients subscribed to that analysis. Internally, the server keeps track of the active analyses in a `ConcurrentHashMap` associating each of them with a `MutableSharedFlow` which represents the stream of status updates for that analysis. When a client subscribes to a specific analysis, the server simply selects the corresponding `Flow` and collects it, sending each status update to the clients as soon as it is emitted as shown in the Listing 3.1 below.

AnalysisRoutes.kt

```kotlin
//SSE analyses/{id}
sse(serialize = { typeInfo, it ->
    val serializer = Json.serializersModule.serializer(typeInfo.kotlinType!!)
    Json.encodeToString(serializer, it)
    }) {
        val id = call.parameters["id"]?.toLongOrNull()
        if (id == null) {
            throw FailureException(
                HttpStatusCode.BadRequest,
                "Invalid or missing analysis id"
            )
        }

        val analysis = analysisManagerService.getAnalysisById(id)
        send(analysis, event = "analysis")

        val flow = analysisManagerService.subscribeToAnalysisEvents(id) ?: return@sse

        flow.collect { analysis ->
            send(analysis, event = "analysis")
        }
}
```

AnalysisManagerService.kt

```kotlin
private val activeAnalyses = ConcurrentHashMap<Long, MutableSharedFlow<Analysis>>()

    fun subscribeToAnalysisEvents(analysisId: Long): Flow<Analysis>? {
        return activeAnalyses[analysisId]
    }
```

Listing 3.1: Server SSE handling

`ConcurrentHashMap` and `MutableSharedFlow` were chosen because they are both thread safe and `Ktor` handles each request in a separate coroutine which means that multiple coroutines may access the same data concurrently. This SSE approach allows to keep clients updated concerning analysis status without the need of polling continuously the server, enabling also the possibility for clients that are not submitting an analysis to receive real time updates of its progress.

Chapter 4

Analysis Workflow

4.1 Analysis Overview

Obviously, the diversity between Android and iOS as platforms and the consequent differences in the application packages require some variations in the analysis approach of Code Guardian depending on the target platform. However, even if the specific analysis tasks performed on `APK` and `IPA` files differ, the overall workflow is similar for both platforms. The tasks performed by the server are categorized in stages, representing a logical grouping of some related analysis tasks. These stages, visually represented in the pipeline of Figure 4.1, are not only useful to categorize and understand the analysis workflow but also because they reflect the current completion status of analyses that are still taking place so that the user can monitor their progress.
Some of the stages depend directly on the results of the right preceding stage while others are only dependent on the package extraction, which sets the baseline strictly needed to further analyse the application.

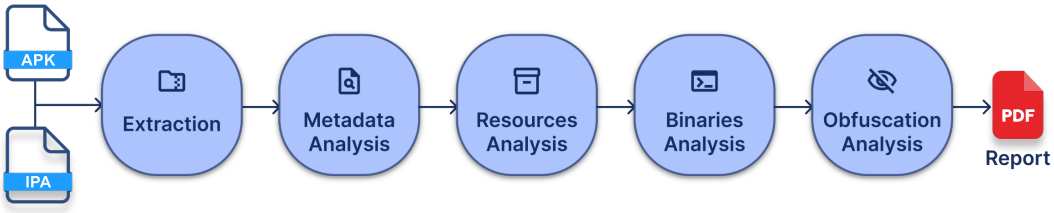

Figure 4.1: Analysis Pipeline

Once a stage is completed, the server stores the results of the analysis and most importantly updates the status of the corresponding analysis. These updates are then reflected in the user interface of clients which requested real-time updates about

that analysis through the SSE based system previously described in Section 3.4. For stages including multiple tasks the server generates also intermediate updates to provide better granularity on the feedback sent to the clients.
The following sections describe in detail each of the stages composing the analysis workflow, outlining meticulously the specific tasks performed by the server.

4.1.1 Security Finds Standardization

The analysis workflow encompasses various security checks and inspections that can have different natures, extending from simple misconfigurations to complex binary protection lack. Some of the issues detected by the server can also have multiple sources, for example, in Android, the clear text network traffic can be enabled both by manifest and by the dedicated netwok configuration file, resulting in the same issue but with different origin and so different recommendations to fix it.
This diversity in nature, origins and recommendations of the security finds is standardized with the Security Finds model represented in Listing 4.1.

SecurityFind.kt

```kotlin
@Serializable
data class SecurityFind(
    val name: String,
    val category: SecurityFindCategory,
    val source: String?,
    val riskLevel: RiskLevel,
    val description: String,
    val recommendation: String? = null,
)
```

Listing 4.1: Security Find Model

Each security find that can be reported by the server is listed in a predefined `JSON` file that contains the description and recommendation for each of them, including also the possible variants based on the origin of the issue. The risk level and category are also predefined in this file so that it has to be specified only once. This model is also used by the client to display the security finds uniformly and also embed them in the final report.

4.2 Package Submission and Extraction

The analysis workflow starts when a user submits an analysis request to the server. The application package to analyse is submitted alongside the request and sent to the server. Once the server receives the package, it stores it in its workspace, discerns the type of package received and accordingly initializes the analysis data

structure. At this time, the analysis is set to the `CREATED` status and the first SSE update is sent to the client in order to notify it about the successful analysis initialization.
Then, the server starts the extraction step of the package. Here there is the first difference between Android and iOS packages analysis. Since `IPA` files are simple `ZIP` archives, the server extracts them using standard `Kotlin` libraries and locates the files of interest for the subsequent stages.
Conversely, `APK` files are extracted leveraging the `apktool`'s extraction and decoding functionalities so that not only the package gets extracted but also manifest and resources files, which are normally in binary format inside the `APK`, are decoded into human readable files.
The just described process is the foundation for all subsequent stages since they all strictly depend on the availability of the extracted files. Once this stage has been completed the analysis status is updated to `EXTRACTED` and the corresponding event is emitted.

4.3 Metadata Analysis

As soon as the package extraction is completed, the server starts the metadata analysis stage. This stage involves several tasks aimed at extracting general information about the inspected package. Although the nature of the information extracted is similar for Android and iOS applications, this stage presents different tasks for the two platforms since the files containing such information are structured differently.

4.3.1 Android Metadata Analysis

Within the Android packages, some configuration parameters about the applications are structured in `XML` files located inside the package. The most important one, which represents the application's identity card is the `AndroidManifest.xml` file, which contains core information and configurations of the app.
After the extraction stage, the `XML` files are already decoded in human readable format so that the server can directly parse them to gather the information that it needs. This parsing is done through the use of `jdom2` which also supports namespaces that are widely used in Android.

General Information

The manifest is analysed to extract first of all general information about the application including:

- Package name and app name
- Version code and name
- Minimum, maximum and target SDK versions
- Supported devices
- App icon

These information are mainly extracted from the `<manifest>` tag and its attributes, along with some other tags like the `<supports-screens>` one. The app icon instead, is located by preferring the main activity icon if present, otherwise falling back to the `<application>` tag icon.
An example of a piece of manifest containing such information from the **InsecureBankv2**[16] package is shown in Listing 4.2. It is visible and noteworthy that in the decoded manifest the resources are not embedded directly but rather referenced through their resource ID like what happens for the app name in the `<application>` tag identified by the `android:label` attribute. This can happen for string resources but also for images and other types of resources. The resolution of these references is done by the server as soon as they are encountered by looking for the corresponding value in the resources files.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1"
    android:versionName="1.0"
    package="com.android.insecurebankv2"
    platformBuildVersionCode="22"
    platformBuildVersionName="5.1.1-1819727">

    <uses-sdk
        android:minSdkVersion="15"
        android:targetSdkVersion="22" />
        <!-- Other content -->
    <application
        android:theme="@android:style/Theme.Holo.Light.DarkActionBar"
        android:label="@string/app_name"
        android:icon="@mipmap/ic_launcher"
        android:debuggable="true"
        android:allowBackup="true">
        <!-- Other content -->
    </application>
</manifest>
```

Listing 4.2: Portion of manifest of InsecureBankv2.apk

Permissions

Moreover, after this general information extraction, the server starts to analyse other important sections of the manifest like the **permissions**, which can also be created by the application itself in Android.
The permissions that are created by the application are extracted by looking for `<permission>` tags and for each of them the server just retrieves their attributes. The requested permissions instead are declared through the `<uses-permission>` tags, each of these tags specifies a single permission requested by the application. The server collects all of them and assigns to each of them a risk level, based on the actions that they allow to perform. This risk level is predetermined with a `JSON` file stored in the server which maps each permission to a risk level, a category and a description. For example, the `INSTALL_PACKAGES` permission, which allows an application to install packages, has a high risk level since it can be exploited to install malicious applications.

Components

The manifest also declares the components composing the application which are activities, services, broadcast receivers and content providers. Each of them is declared with a specific tag but the server most importantly checks if they are exported or not. Components can be exported by explicitly setting the `android:exported` attribute to true or by declaring an `<intent-filter>` tag in the component declaration like shown in the Listing 4.3. Since components exporting is an IPC mechanism and so it represents a possible attack surface for malicious applications, the server checks for both of these conditions signaling them so that these components can be paid particular attention to. Additionally, the current default value of the `android:exported` attribute is false but it changed over time(in particular with Android 4.1.1 the default for content providers became true) and so not specifying it could lead to different behaviours based on the Android version of the device.

```
1  <activity
2      android:label="@string/app_name"
3      android:name="com.android.insecurebankv2.LoginActivity">
4      <intent-filter>
5          <action android:name="android.intent.action.MAIN" />
6          <category android:name="android.intent.category.LAUNCHER" />
7      </intent-filter>
8  </activity>
9      <activity
10     android:label="@string/title_activity_post_login"
11     android:name="com.android.insecurebankv2.PostLogin"
12     android:exported="true"/>
```

Listing 4.3: Exported Activities

Security Checks

The manifest is also the room for some configurations that can directly impact security aspects of the application. For this reason, the server gathers also the associated attributes in order to detect misconfigurations that could lead to weaknesses. In particular, the server analyses the following attributes:

- **Debuggable attribute**
  The `android:debuggable` attribute, visibly set in Listing 4.2, if set to true, allows the application to be debugged even in production. Even if is not strictly a vulnerability, this is obviously a security risk since when this flag is enabled, an attacker could debug a production application and so he could be able to bypass security mechanisms or modify the runtime behaviour of the

app. For this reason, the server checks if this attribute is enabled and reports it.

- **Backup allowed attribute**
  The `android:allowBackup` attribute, also shown in Listing 4.2,if set to true, allows the application data to be backed up using an Android Debug Bridge(**adb**) command. This attribute is set by default to true and so if the developer does not specify it, the application can be backed up.
  Beyond the `android:allowBackup` attribute, if it is enabled there are two more attributes which can be used to restrict the backup scope: `android:fullBackupContent` (Android 11 or lower) and `android:dataExtractionRules` (Android 12 or higher). These attributes point to specific `XML` resource files which basically set the rules for including and excluding files and directories of the backup and starting from Android 9 gives also the possibility to set encryption of those files as required via a flag. Therefore, when enabled, the backup should exclude potentially sensitive files or at least encrypt them.
  The server so checks for the `android:allowBackup` attribute and first of all signals if the backup is allowed. If it is, then the server checks for the encryption requirement flag for the included files and reports if any of them is not required to be encrypted.

- **Test only attribute**
  The `android:testOnly` attribute, when enabled allows the app to be installed only by the means of adb and it might expose functionalities or data that should not be accessible. This attribute is meant to be used only for development and testing so the server checks if it is enabled in the manifest.

Network Configuration

Another important aspect analysed by the server in this stage is the network configuration of the application which spans across multiple files.
Firstly, the manifest can contain the `android:usesClearTextTraffic` attribute of the `<application>` tag which can override the default configuration that since Android 9 enables TLS by default. This attribute can in turn be overridden by a dedicated network security configuration file which can be declared in the `<application>` tag as well through the `android:networkSecurityConfig` attribute. This `XML` file allows to configure settings both at app level by the `<base-config>` tag and at domain level with the `<domain-config>` tag. These tags have an attribute called `cleartextTrafficPermitted` which if set to true can enable the clear text traffic for the associated scope.
The server so inspects this file if present and the `android:usesClearTextTraffic`

attribute signaling if there are misconfigurations allowing clear text traffic because these type of communications represents a security risk.

4.3.2 iOS Metadata Analysis

The core information and configurations of iOS applications are mainly defined in the `Info.plist` file which is in binary format inside the `IPA` package. To read and parse its content properly, the server uses the `ddplist` library which can directly read `plist` files in their binary format.

General Information

Similarly to Android, general information about the application is extracted from the `Info.plist` file including:

- Bundle identifier and app name
- Version name and build
- The SDK name, the minimum OS version and the platform version
- Supported devices
- App icon

These properties are extracted by looking for their corresponding keys in the file such as the `CFBundleIdentifier` key for the bundle identifier and the `DTSDKName` for the SDK name.
In the `plist` file, key-value pairs are organized in dictionaries and arrays with nested structures. For example, the app icon and the supported platforms are located inside nested dictionaries and arrays. This particular composition requires the server to navigate through the nested structures to find the searched information and this is done by a recursive function that explores the entire structure with the `ddplist` library capabilities. An example of a portion of `Info.plist` file from the `iGoat-Swift`[17] package is shown in Listing 4.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<!—— Other Content ——>
  <key>CFBundleExecutable</key>
  <string>iGoat-Swift</string>
  <key>CFBundleIcons</key>
  <dict>
    <key>CFBundlePrimaryIcon</key>
    <dict>
      <key>CFBundleIconFiles</key>
      <array>
        <string>AppIcon29x29</string>
        <string>AppIcon60x60</string>
      </array>
      <key>CFBundleIconName</key>
      <string>AppIcon</string>
    </dict>
  </dict>
  <!—— Other Content ——>
  <key>CFBundleIdentifier</key>
  <string>OWASP.iGoat-Swift</string>
  <!—— Other Content ——>
  <key>CFBundleName</key>
  <string>iGoat-Swift</string>
  <!—— Other Content ——>
  <key>MinimumOSVersion</key>
  <string>10.0</string>
  <!—— Other Content ——>
</dict>
</plist>
```

Listing 4.4: Portion of `Info.plist` of iGoat-Swift.ipa

Permissions

The `Info.plist` file is also the place where iOS applications declare the permissions that they require to provide specific features.
Each permission requestable by an application has a corresponding key that the app has to include in its `Info.plist` file in order to request it. The permission key is accompanied by a rationale string, which became mandatory starting from iOS 10 and that will be shown to the user when the permission is requested at runtime. Therefore, the server gathers all the permissions requested by the application and assigns them a risk level with the same strategy that is used for the Android permissions, based on a predefined `JSON` file. The permissions made available by iOS are not as many as the ones in Android but they still cover sensitive resources like camera, microphone and location.

App Transport Security Configuration

As explained in Section 2.4.6, iOS applications' network security is enforced by the ATS policies by default. However, these policies can be relaxed by configuring the

`NSAppTransportSecurity` key in the `Info.plist` file and so the server inspects it to evaluate the network configuration of the application.
The `NSAppTransportSecurity` key identifies a dictionary that can contain both global and domain specific exceptions to the ATS restrictions, it is shown in Listing 4.5 the structure of this dictionary and its keys. The boolean keys are default set to false("NO") except for the `NSExceptionRequiresForwardSecrecy`.

```
NSAppTransportSecurity : Dictionary {
    NSAllowsArbitraryLoads : Boolean
    NSAllowsArbitraryLoadsForMedia : Boolean
    NSAllowsArbitraryLoadsInWebContent : Boolean
    NSAllowsLocalNetworking : Boolean
    NSExceptionDomains : Dictionary {
        <domain-name-string> : Dictionary {
            NSIncludesSubdomains : Boolean
            NSExceptionAllowsInsecureHTTPLoads : Boolean
            NSExceptionMinimumTLSVersion : String
            NSExceptionRequiresForwardSecrecy : Boolean   // Default value is YES
            NSRequiresCertificateTransparency : Boolean
        }
    }
}
```

Listing 4.5: ATS dictionary structure

`NSAllowsLocalNetworking`, `NSAllowsArbitraryLoads` and its variants are globally scoped exceptions. The first one, if set, allows connections to local domains. Conversely, the second key and its variants disable the ATS restrictions, the main one in general and the variants for specific connection types.
The `NSExceptionDomains` key instead identifies the dictionary where domain specific exceptions can be declared. Within this dictionary there is the possibility to relax only some of the ATS restrictions for each domain by setting the corresponding keys, in particular:

- **`NSIncludesSubdomains`**
  This key sets if the exceptions are applied also to the subdomains of the specified domain.

- **`NSExceptionAllowsInsecureHTTPLoads`**
  This key if set to true allows HTTP connections to the specified domain but does not affect the TLS requirements for HTTPS connections.

- **`NSExceptionMinimumTLSVersion`**
  This key is used to decrease the minimum required TLS version to versions lower than 1.2 for the specified domain.

- `NSExceptionRequiresForwardSecrecy`
  This key, which is the only one default set to true, can be used to disable the PFS requirement for the specified domain.

- `NSRequiresCertificateTransparency`
  This key is obsolete, it was used to indicate if the app required Certificate Transparency(**CT**) for the specified domain. CT requires that the server's certificates to have support from signed CT timestamps from at least two different CT logs trusted by Apple. Now the system requires CT by default and so this key has no effect.

The server so evaluates these configurations, reporting the presence of possibly insecure settings. Nevertheless, it should be considered that some applications like browsers may require these exceptions to fullfill their purpose and so some of the applied exceptions may be acceptable. Since the focus of Code Guardian is on banking applications, these exceptions have to be carefully analysed because they should not be strictly necessary in this domain.

4.4 Resources Analysis

The second stage of the analysis pipeline is the resources analysis. The focus of the resources analysis is to inspect string resources defined in the packages in order to detect the presence of hardcoded secrets like API keys, passwords or cryptographic keys. Even if this is a bad practice that should be avoided, it is still extensively present in mobile applications as testified by OWASP Mobile Top 10 where "M1: Improper Credential Usage" is the most critical risk for mobile applications and it includes hardcoded credentials. Also data leakage and hardcoded secrets in general are considered by OWASP as a serious issue even if they did not make the place in the top 10 list.
Even if resources are included in distinct ways in Android and iOS packaged, once they have been extracted, they can be treated in the same way to achieve the purpose of this stage so the analysis tasks are the same for both platforms, the difference lies only on how the resources are located and gathered.

4.4.1 Strings Analysis

String resources are typically used to define user visible texts of the application interface like labels, buttons, placeholders and so on in a centralized way. Despite this is their main purpose, they can also be improperly used to store other types of strings like URLs, API keys or any other string that the application may use and that can be defined outside of the source code. They are easily accessible

by extracting the package and so they represent a weak spot where sensitive information may be exposed if not carefully monitored.
Moreover, in mobile applications, textual resources are usually localized. This means that the same string resource can have multiple variants which will be the translations of the string for the different languages supported by the application. The main variant is usually in English and it serves as a fallback when the localized variant for some language is not defined.
In Android, the package contains string resources in the `resources.arsc` file in binary format but the `apktool` extraction process allows to retrieve them as they are at development time, in human readable `XML` files located in the `res/values` directory or in the localized versions of this directory like `res/values-it` for Italian. The strings are then organized in `<string>` tags inside the `XML` files, having a key represented by the name attribute and the actual string as the content of the tag. Since this representation in `XML` is the same of the manifest and other configuration files, the server adopts the same parsing strategy based on `jdom2` to extract the strings.
In iOS packages the organization is nearly the same even if the file format is different, string resources are stored in `.strings` files in a binary plist format. These files are located in `.lproj` directories with each language having its dedicated folder and the `Base.lproj` directory for the default variant. In these files the strings are organized as key-value pairs where the key is the identifier of the string and the value is the actual string. Also in this case the server uses the same approach used to parse the `Info.plist` file with the `ddplist` library to extract the strings since they are in binary `plist` format as well.
For both platforms, the strings extraction brings as side effect to also get the supported localizations of the application. They are used to inspect if the application has some missing string translation for any of the supported languages providing also a useful quality metric about the application.

Secrets detection

Once all the strings have been extracted from the package, the server starts the secrets detection process.
This process was initially based on rules containing exclusively regex patterns to match common formats of secrets. The regex patterns were partially retrieved by `gitleaks`, supplemented with some additions related to the banking domain like patterns for credit card numbers or IBANs, and organized in predefined rules in a `JSON` file. The strings were just matched against the rules' patterns and if there was a match the string was considered a potential hardcoded secret.
Nevertheless, this approach has shown some limitations in terms of accuracy since matching strings against regexes can lead to a lot of false positives because it does

not take in consideration how the string is conformed and that some particular strings that match the patterns are not actual secrets. Therefore, to improve the accuracy in the detection process, it has been integrated a system to exclude some of the matched strings based on stop-words, stop-regexes and **Shannon entropy** thresholds. Each rule, to enhance the detection accuracy, can optionally include these additional filters which are applied if the string matched the rule's regex pattern.
Stop-words and stop-regexes are used respectively to exclude strings that contain specific words or match specific patterns. For example, a string containing the word *example* or *placeholder* is probably not a secret even if it matches some rule's pattern, so it can be discarded.
Furthermore, the Shannon entropy is a metric that measures the randomness of a string based on the frequency of its characters. The higher the entropy, the more random the string is. Given a string S and its length L, its entropy is calculated as:

$$H(S) = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i)$$

where n is the number of unique characters in the string and $P(x_i)$ is the probability of occurrence of character x_i in the string, computed as:

$$P(x_i) = \frac{c_i}{L}$$

where c_i is the number of occurrences of the character x_i in the string S.
Each rule can so be filtered by a minimum entropy threshold that the string must exceed to be considered a potential secret and a maximum entropy threshold that the string must not exceed. This precise filter is really effective and valuable since secrets like API keys have a medium-high entropy and others like credit card numbers or IBANs are not completely random so they also need a maximum threshold since there are some fixed patterns. An only regex based approach could for example match a string like `1111 2222 3333 4444` as a possible credit card number but clearly this number is not valid. Its entropy is 2.0 which is low for a credit card number so it can be discarded.

4.5 Binaries Analysis

Metadata and resources analysis provide important information about configurations and possible hardcoded secrets but the core of an application lies in its binary files since they contain the actual code that is executed on the device. This stage of the pipeline shifts the focus from the configuration files and resources to the application binaries and examines them in order to ensure that they accurately

implement some security mechanisms and also sets the basis for the subsequent obfuscation analysis stage. The scope of this stage covers not only the main executables but also the libraries and frameworks included in the package.
Binary files contained within the mobile applications packages are of various types, spanning from unique formats to more common ones. In `IPA` packages, the main executable, libraries and frameworks included in the package are in `Mach-O` format. Conversely, in `APK` files there are `DEX` files containing the Dalvik bytecode which represent the main executables and there can also be native libraries in `ELF` format. For both platforms, the native libraries binaries are usually repeated for multiple architectures and organized in dedicated directories to ensure compatibility across different devices.
However, they mostly share the same elements like strings and symbols even if they are organized differently in the various formats. The `DEX` format is the one that differs the most from the others since it is bytecode and not machine code like `Mach-O` and `ELF` files but it still shares some characteristics with them.

4.5.1 Ghidra

The just mentioned heterogeneity of formats is ruled by **`Ghidra`** which represents the backbone of this stage. Its comprehensive support for all these formats and lots of architectures makes it the perfect tool to fullfill the requirements of this stage.
`Ghidra` functionalities are based on projects that can contain multiple binaries to analyse, in particular a binary is called `Program` in `Ghidra` terminology. The binaries can be imported in a project and once imported `Ghidra` automatically analyses them with a set of predefined options that can be customized. The auto analysis of `Ghidra` involves at minimum starting by the entry points of the binary and disassembling by following the flows. It also allows to identify functions, strings and symbols. In general, `Ghidra`'s features are accessible by its interactive Graphical User Interface(GUI) but since Code Guardian's analysis has to be automatic its headless version has been preferred. **`Headless Ghidra`** allows to run `Ghidra`'s auto analysis and scripts through a CLI. The scripts are writable in `Java` or `Python` using the `Ghidra` API but since Code Guardian's main language is `Kotlin`, they have been written in `Java` to be as consistent as possible with the server's codebase. Therefore, Code Guardian's server employs `Headless Ghidra` by containerizing it and invoking it when needed.
Being open source, the `Ghidra` container's image is built from the official `Ghidra` repository and customized by mounting as volume, besides input and output folders, the folder containing the scripts. With this setup the server can easily invoke `Headless Ghidra` by just specifying the input package (or directly binaries) and the script to run. The scripts have output data that is stored in the output folder in Newline Delimited JSON(`NDJSON`) format which has been chosen instead of the

standard JSON because Headless Ghidra runs the scripts independently for each binary and so it is more convenient to build the output file just appending each binary's output as a new line in the NDJSON rather than having to rewrite the entire object or array structure of JSON for each binary.
The first operation done by the server in this pipeline stage is creating the Ghidra project and importing the package in it with the analysis disabled, so that the Ghidra auto analysis is executed only once the environment is fully setup. After that, the server requests Headless Ghidra to perform its auto analysis and then execute specific scripts to perform the tasks of this stage. In this way, Ghidra's auto analysis is performed by taking into account the entire context of the package, not only on single binaries and so the results are more accurate. This actually has no advantages in iOS since libraries are anyway imported at runtime but in Android it allows multiple DEX files to be analysed knowing the presence of each other and so cross-references between them can be properly identified.
Moreover, after the auto analysis, the requested scripts can access and take advantage of its results as described in the following sections.

4.5.2 Strings and Symbols Extraction

The first Ghidra scripts that are executed are the ones to extract strings and symbols from the binaries. The strings extraction is clear-cut since they are by default identified by Ghidra's auto analysis and are accessible via the **DefinedStringIterator** class and the general **Data** interface of the Ghidra API. DefinedStringIterator allows to iterate along all the strings identified in the binary while the Data interface serves as a general representation of data at a certain address in the binary. Therefore, as shown in the Listing 4.6, for each binary of the package the script iterates through all the strings collecting their address, type and actual string value to build the output NDJSON object. The value of the string is also sanitized by removing non-printable characters since it can happen that some of the strings identified by Ghidra contain such characters.

ExtractStrings.java

```java
public List<String> extractStrings(Program program) throws Exception {
        List<String> extractedStrings = new ArrayList<String>();

        String name = program.getName();
        String executablePath = program.getDomainFile().getPathname();
        executablePath.replace("/home/ghidrauser/output", "server/src/main/workspace/packageExtraction");

        DefinedStringIterator iter = DefinedStringIterator.forProgram(program);
        JsonWriter writer = new JsonWriter(new FileWriter("./analysis/ghidra/ghidraStrings.ndjson", true));
        writer.setLenient(true);
        writer.setIndent("");

        writer.beginObject();
        writer.name("program").value(executablePath);
        writer.name("strings");
        writer.beginArray();

        while (iter.hasNext()) {
            Data data = iter.next();
            String value = sanitizeNdjsonField(data.getValue().toString());
            if(value!="") {
                String address = data.getAddress().toString();
                String dataTypeName = data.getDataType().getName();

                extractedStrings.add(value);
                writer.beginObject();
                writer.name("address").value(address);
                writer.name("string").value(value);
                writer.name("type").value(dataTypeName);
                writer.endObject();
            }
        }
        writer.endArray();
        writer.endObject();
        writer.flush();
        writer.close();
        // Manually append newline since JsonWriter doesn't
        FileWriter newlineWriter = new FileWriter("./analysis/ghidra/ghidraStrings.ndjson", true);
        newlineWriter.write("\n");
        newlineWriter.close();
        return extractedStrings;
    }
```

Listing 4.6: `Ghidra` Strings Extraction

The symbols extraction instead is slightly more complex since it includes both regular symbols and debug symbols. Regular symbols extraction is handled with a strategy similar to the strings one, the difference is that the **`SymbolIterator`** class is used to iterate in place of the `DefinedStringIterator` and the **`Symbol`** interface is used instead of the `Data` one. In addition to address, type and name, for each symbol the script also gathers its namespace which can be useful to understand the context of the symbol.
The debug symbols extraction instead is done by the means of the `objdump --syms` command, which is suggested by the OWASP MASTG as a possibility to assess the presence of debug symbols. In particular, the server executes the `objdump --syms` command for each binary that is not a `DEX` file and then parses the output filtering only the symbols having the **d** flag which indicates that they are debug symbols.

Precisely, for the `Mach-O` files, the server discards the symbols `radr://5614542` and `__mh_execute_header` since they are not relevant debug symbols.

4.5.3 Binary Header Information and Protections

Right after the strings and symbols extraction, the third script executed by `Headless Ghidra` is the one responsible for extracting some information about the headers and security protections of the binaries, reporting if any of the protections is absent. The security protections checks are implemented emulating the logic of `LIEF`, which is a library for parsing executable formats available in `Python`, `C++` and `Rust`. Since `LIEF` is not available in `Java`, the script reproduces the same verifications by using the `Ghidra` API.
The strings and symbols extracted previously are propagated as input to this script so that it can use them when needed. Additionally, Even in this instance, `DEX` files are excluded and the scripts for `ELF` and `Mach-O` files are completely different due to their distinct structures.

ELF

For `ELF` files, the script uses the `ElfHeader` class of the `Ghidra` API, which abstracts the `ELF` header structure shown in Listing 4.7.

```
#define EI_NIDENT 16

//32 bit
typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;

//64 bit
typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf64_Half      e_type;
        Elf64_Half      e_machine;
        Elf64_Word      e_version;
        Elf64_Addr      e_entry;
        Elf64_Off       e_phoff;
        Elf64_Off       e_shoff;
        Elf64_Word      e_flags;
        Elf64_Half      e_ehsize;
        Elf64_Half      e_phentsize;
        Elf64_Half      e_phnum;
        Elf64_Half      e_shentsize;
        Elf64_Half      e_shnum;
        Elf64_Half      e_shstrndx;
} Elf64_Ehdr;
```

Listing 4.7: `ELF` header[18]

By using this class, the script first of all gathers the binary endianness and identifies if it is 32 or 64 bits. Then, it starts to check for the presence of the following security protections:

- **NX**
  The NX presence is verified by firstly locating the program header table and then iterating through its entries looking for the `PT_GNU_STACK` segment. If it is not present, the NX protection is considered disabled, otherwise the script checks the flags of this segment to see if the execute flag is set with a bitwise AND operation.

- **PIE**
  The PIE usage is checked with the `isRelocatable` and `isSharedObject`

methods of the `ElfHeader` class which specifically check if the `e_type` field of the header is set to `ET_REL`(1) or `ET_DYN`(3) respectively.

- **Stack Canaries**
  Stack canaries check exploits the symbols extraction performed before and inspects for the presence of `__stack_chk_fail` or `__intel_security_cookie` symbols which are indicative of canaries usage.

Mach-O

On the other hand, `Mach-O` files are analysed in a mirrored way, using the `MachHeader` class which abstracts the `Mach-O` header structure shown in Listing 4.8.

```
1  //32 bit
2  struct mach_header {
3     uint32_t    magic;      /* mach magic number identifier */
4     cpu_type_t    cputype;    /* cpu specifier */
5     cpu_subtype_t    cpusubtype;    /* machine specifier */
6     uint32_t    filetype;    /* type of file */
7     uint32_t    ncmds;      /* number of load commands */
8     uint32_t    sizeofcmds;    /* the size of all the load commands */
9     uint32_t    flags;      /* flags */
10 };
11
12 //64 bit
13 struct mach_header_64 {
14    uint32_t    magic;      /* mach magic number identifier */
15    cpu_type_t    cputype;    /* cpu specifier */
16    cpu_subtype_t    cpusubtype;    /* machine specifier */
17    uint32_t    filetype;    /* type of file */
18    uint32_t    ncmds;      /* number of load commands */
19    uint32_t    sizeofcmds;    /* the size of all the load commands */
20    uint32_t    flags;      /* flags */
21    uint32_t    reserved;    /* reserved */
22 };
```

Listing 4.8: `Mach-O` header[19]

As in the `ELF` case, the script retrieves the endianness and bitness as first step. Then, it gathers the flags of the header because they are needed to perform the majority of the following verifications:

- **NX and PIE**
  The flags of the header are examined in order to check if the flags `MH_PIE`

and `MH_ALLOW_STACK_EXECUTION` are set or not. If the first is set, the PIE protection is enabled, otherwise it is disabled. Conversely, if the second flag is set, the NX protection is disabled, otherwise it is enabled.

- **Stack Canaries**
  The canaries check is performed similarly to the `ELF` case by looking for `___stack_chk_fail` and `___stack_chk_guard` symbols within the previously extracted symbols. The canaries are enabled if both of them are present.

4.5.4 Functions Extraction

The last `Ghidra` script executed before the decompilation is the one for functions characteristics extraction. The functions are analysed in their disassembled form and so it has to be considered that the results may not be as accurate as if they were analysed in source code because the disassembling process is not deterministic and precise. Additionally, the number of instructions in disassembled functions is obviously different from the number of instructions in source code because a high level instruction can result in multiple low level instructions in assembly. Another aspect to take into account is that assembly instructions are architecture dependent but in this specific environment it does not represent a problem since `Ghidra` abstracts the instruction sets with a unique representation called **P-Code** which is architecture independent. A single assembly instruction can be translated in one or more P-Code Operations (`PCodeOp`) that are used by this script to analyse the instructions of the functions.
The script exploits the `FunctionManager` class to iterate through the functions of the binary and for each of them it collects some of their features. First of all, the script gathers their cyclomatic complexity using the dedicated method.
Afterwards, it analyses each `CodeBlock` of the function and in particular their possible destinations in order to detect eventually dead blocks, which are simply the blocks that are not destinations of any other block. Thus, these blocks and their instructions are marked as dead. In these phase also trivial instructions are detected and marked as dead with a simple logic that considers as trivial instructions operations like identity assignments, additions or subtractions with zero and multiplications or divisions by one.
Then, the script employs a powerful feature of `Ghidra` which is the `Symbolic Propogator` class. This class is used to perform constant propagation on the P-Code operations of the function. This technique allows to track the values of variables along the function's flow and possibly identify the ones which assume always the same value, marking them as constants. This way, the script can identify the opaque predicates that are based on conditions involving only constants values and so update the dead blocks and instructions accordingly to the always or never taken branches.

This approach is effective in identifying some of the opaque predicates but it has some natural limitations since it can only track values explicitly defined in the function and moreover it fails in identifying predicates that are clearly opaque but not based on constant values like for example `if((x*0)==0)` when `x` is an input variable.
The last task performed by the script is to detect if the function has possibly been obfuscated with control flow flattening. As shown in Section 2.4.10, the control flow flattening is characterized by the presence of a dispatcher block that manages the flow of the function typically through a `switch` statement. Thus, in order to flag the function as flattened, the script inspects the single blocks looking for the presence of such dispatcher block also because in natural functions this kind of structure is basically an anti patterns and so it is rare to find it in non obfuscated functions. To achieve this purpose, each block is analysed firstly checking the number of its predecessors and successors since the dispatcher typically has multiple blocks that can jump to it but also multiple possible destinations. If the block has a high number of predecessors and successors relative to the total function size (>20%), it is furtherly analysed scanning its `PCodeOp`s to find a `switch` statement. Specifically, the script looks for `PCodeOp`s of type `BRANCHIND` and `CBRANCH` which are indicative of indirect and conditional branches respectively. If one of these `PCodeOp`s is found, the function is marked as possibly flattened.
Finally, the script builds the output `NDJSON` file containing for each function its name, cyclomatic complexity, total and dead instruction counts, reachable and dead blocks counts, the number of always or never taken branches and the flag for control flow flattening.

4.5.5 Decompilation

The final operation performed by `Ghidra` is its peak feature, the decompilation. It is done by using the `DecompInterface` class of the `Ghidra` API which includes the methods to decompile functions and retrieve the results. The script is quite direct since it just iterates through all the functions, decompiles them and collects their decompiled code. The noteworthy aspect of the decompilation is that it is really demanding in terms of resources and time. A single function can also take several seconds to be decompiled and so for large binaries containing great number of functions, this process can take a lot of time.

4.5.6 Semgrep Rules Application

The decompiled code produced in the previous task by `Ghidra` is then analysed by the means of **`Semgrep`** which allows to define a set of custom rules to search for specific functions and patterns in the code. The usage of `Semgrep` comes

with the goal of analyse beyond the simple presence of binary protections and obfuscation, examining in detail the actual decompilation output. For example, it has been defined a rule to check for the presence of insecure symmetric cryptographic algorithms in `Cipher`. The applied rule is the one shown below in Listing 4.9.

rules.yml

```yaml
rules:
  - id: InsecureCipherCrypto
    languages:
      - c
      - cpp
    severity: WARNING
    message: Insecure Cipher Crypto
    patterns:
      - pattern: $FUNC(..., "$ALG", ...)
      - metavariable-regex:
          metavariable: $FUNC
          regex: (?i).*Cipher.*(::|\.|_)getInstance$
      - metavariable-regex:
          metavariable: $ALG
          regex: ^(DES|DESede|RC4|Blowfish)(/[A-Za-z0-9]+(/[A-Za-z0-9]+)?)?$
```

Listing 4.9: Semgrep Insecure Cipher Rule

The problem in rules building is that the decompiled code of `Ghidra` is in a pseudo-C language so the rules have to be defined accordingly and it is not always immediate to identify the right patterns to match. For example, the just shown rule looks for the `Cipher.getInstance()` method called with for example `Blowfish` as parameter. Nevertheless, it is not directly searched as it is but instead the rule matches also other structures like `Cipher::getInstance()`. This is necessary because the decompiled code extracted by `Ghidra` may contain different representations of method calls.
Additionally, the real downside of this strategy is that with obfuscated code the effectiveness of the rules is compromised since symbols obfuscation can alter names of classes and methods resulting in the rules failing to match them.

4.6 Obfuscation Analysis

The final stage of the analysis pipeline is the obfuscation analysis. The main goal of this stage is to assess if obfuscation has been applied to the binaries and to which extent, assigning an overall obfuscation score and several sub-scores related to specific obfuscation techniques and single binaries. These scores so provide a quantitative and fine-grained metric to analysers that can be useful to identify

where the obfuscation process is lacking and can be improved.
The strategy adopted by the server to achieve this purpose is based on examining the features of strings, symbols and functions extracted by `Ghidra` in the previous phase, in order to detect if some obfuscation techniques have been applied. The techniques that are taken into account are described in Section 2.4.10 and grouped in three categories: strings obfuscation, symbols obfuscation and functions obfuscation. The latter one includes both dead code injection and control flow obfuscation.
Therefore, for each binary, the server inspects every string, symbol and function classifying them as obfuscated or not based on their features.
These classifications are then used to compute the scores of each technique as well as the overall score of the binary. Finally, the scores of all the binaries are aggregated to compute the obfuscation scores of the entire application, both overall and per technique. This whole process is explained in detail in the next sections.

4.6.1 Strings and Symbols Classification

Strings and symbols represent different elements of the binaries but at the end of the day they are both textual data and therefore they share the same features. As a consequence, even if the obfuscations techniques applied on them are different, the classification rationale adopted for both of them is based on the same features which are in the following Listing 4.10 and later on referred as $\phi(S)$.

SymbolFeatures.kt

```kotlin
@Serializable
data class SymbolFeatures(
    val hasSpecialChars: Boolean,
    val hasSemanticMeaning: Boolean,
    val entropy: Double,
)
```

StringFeatures.kt

```kotlin
@Serializable
data class StringFeatures(
    val hasSpecialChars: Boolean,
    val hasSemanticMeaning: Boolean,
    val entropy: Double,
)
```

Listing 4.10: Strings and Symbols Features

Firstly, the length was considered viable as feature since one of the strategies commonly adopted by obfuscators is to subsitute symbols with one single character or very short names. It could be a good indicator to detect this type of obfuscation but at the same time it would have been useless for other techniques. Therefore, the length has been substituted by an **entropy** based feature, called $\phi_E(S)$, because the entropy is a more powerful metric that can fit more obfuscation techniques. Specifically, the entropy of the string or symbol is checked against a predefined non-obfuscated range in order to determine whether it is obfuscated or not as

follows:

$$\phi_E(S) = \begin{cases} 1 & H(S) < H_{\min} \;||\; H(S) > H_{\max} \\ 0 & \text{Otherwise} \end{cases}$$

where $H(S)$ is the entropy of the string or symbol S and $H_{\min}$ and $H_{\max}$ are the minimum and maximum entropy values of the non-obfuscated range respectively. These bounds have been determined empirically as $H_{\min} = 2.5$ and $H_{\max} = 4.5$. In particular, if the entropy falls outside of these bounds, according to this feature $\phi_E(S)$, the string or symbol should be considered as obfuscated, otherwise not obfuscated.

This feature in place of length results more versatile since it can detect both very short strings or symbols like a symbol named *a* that has 0 entropy but also strings or symbols with high entropy that are made of random characters like possibly encrypted strings.

The second feature is the presence of special characters in the string or symbol indicated with $\phi_{SC}(S)$. Special characters are rare in regular strings and even more in symbols since they usually include only alphanumerical characters so their presence can be a good indicator of obfuscation.

Lastly, the most important feature is the semantical meaning indicated with $\phi_M(S)$. Regular strings included in binaries are typically used for things like labels, placeholders and error messages so they are built to be meaningful. On the other hand, symbols are identifiers, variables, class names, function names and so they also are meant to be meaningful. Therefore, this feature is the most significant one because the removal of the inherent semantical meaning of strings and symbols is the ultimate goal of obfuscators in this context.

The evaluation of this feature is considerably complex because it requires a level of comprehension that is challenging to achieve automatically. This difficulty is further amplified because strings and symbols can appear in multiple languages, abbreviations and acronyms which complicates to find a universal approach. Additionally, their meaning can heavily depend on the context. For example, the string *"OWASP"* is meaningful in cybersecurity domain but it could be pointless in other contexts.

To tackle these challenge, the server employs a **Large Language Model** hosted on Azure AI Studio and accessed through its API. The chosen model is GPT 4.1 and it is queried with a carefully designed prompt and an explicit `JSON` schema to request a structured response, which can then be easily parsed by the server.

```
You are an assistant that analyzes strings and symbols from mobile banking applications'
binaries for meaningful content in any language, abbreviation, acronym and case(camelCase,
snake_case ecc) and for Base64 decoding.
Analyze the following string: INPUT_STRING_OR_SYMBOL
Follow these rules carefully:
1. Check if the input string has semantical meaning in any language, abbreviation, acronym
and case(camelCase, snake_case ecc).
   - If it does, set "semanticalMeaning": true.
   - If it does not, set "semanticalMeaning": false.
2. Only if the input string does not have semantical meaning, check if it is valid Base64.
   - If it is valid Base64, decode it.
   - If decoding was performed, set "performedBase64Decoding": true otherwise set
   "performedBase64Decoding": false.
3. Only if decoding was performed, check if the decoded string has semantical meaning
   in any language, abbreviation, acronym and case(camelCase, snake_case ecc).
   - If it does, overwrite "semanticalMeaning" to true.
   - If it does not, leave "semanticalMeaning" as false.
4. If the input string was not valid Base64, set "performedBase64Decoding": false.
```

Listing 4.11: LLM Prompt for Strings and Symbols Meaningfulness Evaluation

The prompt, presented in Listing 4.11, is designed to provide the model with the context of mobile banking applications and instruct it to evaluate the meaningfulness of the input for all the possible languages, abbreviations and case variations like camel case and snake case that are typically found in strings and symbols. In particular, the prompt asks the model to determine if the input is meaningful or not and, if not meaningful, the model is requested to furthermore assess whether the `Base64` decoded version of the input is meaningful. This additional examination is performed because `Base64` encoding is a weak obfuscation technique that can be easily reversed, so if the decoded version is meaningful, the string or symbol has only been superficially obfuscated, resulting in a lower score compared to inputs that hide their real meaning even after decoding them. The instructions given by the prompt indicate also how to precisely fill the attributes of the object defined as structured response in the `JSON` schema.
The schema includes two attributes: `semanticalMeaning` which is a boolean indicating if the input is meaningful or not and `performedBase64Decoding` which is a boolean as well, indicating if the `Base64` decoding has been performed. These attributes, referred in the formula below respectively as M and B, are so used by the server to understand if the string or symbol is obfuscated, obfuscated by only `Base64` encoding or not obfuscated at all, and assign the corresponding value to the semantical meaning feature $\phi_M(S)$ as follows:

$$\phi_M(S) = \begin{cases} 1 & M = \text{false} \;\&\&\; B = \text{true} \\ 0.5 & M = \text{true} \;\&\&\; B = \text{true} \\ 0 & M = \text{true} \;\&\&\; B = \text{false} \end{cases}$$

Note that the case $M = \text{false}$ && $B = \text{false}$ is not possible according to the prompt instructions considering that if the semantical meaning is not found in the normal input the `Base64` decoding must be always performed. Once all the three features have been evaluated, the server classifies the string or symbol by the means of a weighted sum of the features compared against a threshold as follows:

$$O(S) = \sum_{i \in \{E,SC,M\}} (w_i \cdot \phi_i(S))$$

where $O(S)$ represents the final obfuscation score given to the symbol or string and the three weights of the sum are empirically defined as:

$$w_E = 0.3 \quad w_{SC} = 0.2 \quad w_M = 0.5$$

Each feature is assigned a weight based on its relevance and strength in obfuscation detection. The entropy based feature has a moderate weight since it is useful but not sufficient alone to determine obfuscation, the special characters feature has the lowest weight because it is just a supporting feature less reliable than the others and at last the semantical meaning feature has the highest weight because it is the most relevant one.
Finally, since all the values involved in the computation are normalized between 0 and 1 and the weights summed together equal 1, the score $O(S)$ will also be in the range $[0, 1]$ and so each string or symbol is classified as obfuscated if the score $O(S)$ is greater than or equal to the threshold $\tau = 0.5$.

4.6.2 Functions Classification

Functions are more byzantine elements compared to strings and symbols, they are not just a textual element but rather code blocks and so their classification requires a different approach. The classification performed by Code Guardian is referred to the functions' bodies disassembled by `Ghidra`'s auto analysis and to their characteristics extracted in the previous phase. The features adopted for this classification are listed below in Listing 4.12 and afterwards referred as $\phi(F)$.

FunctionFeatures.kt

```kotlin
@Serializable
data class FunctionFeatures(
    val deadCodeRatio: Double,
    val opaquePredicatesRatio: Double,
    val possiblyFlattened: Boolean,
)
```

Listing 4.12: Functions Features

The first two features, dead code ratio($\phi_{DC}(F)$) and opaque predicates ratio($\phi_{OP}(F)$) are just computed by dividing the number of dead instructions by the total number of instructions and dividing the number of opaque predicates by the total number of branches respectively.

$$\phi_{DC}(F) = \frac{I_{dead}(F)}{I_{tot}(F)} \cdot 100 \qquad \phi_{OP}(F) = \frac{B_{op}(F)}{B_{tot}(F)} \cdot 100$$

where I represents instructions and B branches. These two features are important but not as decisive as the possibly flattened control flow feature($\phi_{CF}(F)$) because the simple presence of dead code and opaque predicates can also be just a matter of chance or bad coding practices, while the flattened control flow is a more decisive indicator of obfuscation. It is still not flawless since dispatchers can also be implemented in non obfuscated functions but it is a much rarer case. Therefore, the classification is performed also in this case with a weighted sum of the features as follows:

$$O(F) = \sum_{i \in \{DC,OP,CF\}} (w_i \cdot \phi_i(F))$$

where $O(F)$ represents the final obfuscation score and the three weights of the sum are empirically defined as:

$$w_{DC} = 0.2 \quad w_{OP} = 0.3 \quad w_{CF} = 0.5$$

As in the strings and symbols case this score is then checked against a threshold $\tau = 0.5$ to finally classify the function as obfuscated or not.

4.6.3 Scores Computation

As soon as all the strings, symbols and functions of each binary have been classified as obfuscated or not, the obfuscation analysis proceeds to compute the subsequent aggregated scores. The first task is to compute the scores of each obfuscation technique for each binary, these scores are computed as a simple ratio between the number of obfuscated elements and the total number of that type of elements in the binary and they are expressed in the range $[0, 100]$. The only exception is given by the symbols obfuscation score which is computed considering that the eventual presence of debug symbols negatively affects the score since they should be stripped.

$$O_{ST}(B) = \frac{ST_{obf}(B)}{ST_{tot}(B)} \cdot 100$$

$$O_{SY}(B) = \left(\frac{SY_{obf}(B)}{SY_{tot}(B)} - (SY_{debug}(B) \cdot \Delta) \right) \cdot 100$$

$$O_{FU}(B) = \frac{FU_{obf}(B)}{FU_{tot}(B)} \cdot 100$$

where $O_{ST}(B)$, $O_{SY}(B)$ and $O_{FU}(B)$ are respectively the strings, symbols and functions obfuscation scores of the binary B. Δ is the penalty applied for the presence of debug symbols and it has been set to 0.15.
Once the scores of each obfuscation technique for a binary have been computed, the overall obfuscation score of the binary is calculated as a weighted sum of the three technique scores in this way:

$$O(B) = \sum_{i \in \{ST,SY,FU\}} (w_i \cdot O_i(B))$$

where the weights have been empirically defined as:

$$w_{ST} = 0.2 \quad w_{SY} = 0.4 \quad w_{FU} = 0.4$$

However all of the three techniques are important to ensure a proper obfuscation but in order to hide the logic of an application, its goals, its flows and its most sensitive parts, the obfuscation of function bodies and symbols is more decisive than strings obfuscation, which is less impactful since it focuses on textual strings that may not be strictly necessary to disguise the core of the application. Therefore, symbols and functions obfuscation scores have been assigned higher weights than the strings one.
Finally, after having completed the calculation of the scores for each binary, the last step of this stage is to furtherly aggregate the scores of all the binaries to compute a score for each technique and an overall obfuscation score for the entire application. The computation of the scores of techniques is performed similarly to the single binary case but in this case the scores of each binary are weighted by the number of elements of that type that the binary contains.

$$O_{ST} = \frac{\sum_{B \in \mathcal{B}} (O_{ST}(B) \cdot ST_{tot}(B))}{\sum_{B \in \mathcal{B}} ST_{tot}(B)}$$

$$O_{SY} = \frac{\sum_{B \in \mathcal{B}} (O_{SY}(B) \cdot SY_{tot}(B))}{\sum_{B \in \mathcal{B}} SY_{tot}(B)}$$

$$O_{FU} = \frac{\sum_{B \in \mathcal{B}} (O_{FU}(B) \cdot FU_{tot}(B))}{\sum_{B \in \mathcal{B}} FU_{tot}(B)}$$

where $\mathcal{B}$ is the set of all the binaries of the application. Clearly, since the number of obfuscated elements of each type represent the weight of each binary's score and it is not normalized, the final weighted sums are normalized by the total number of elements of that type in the entire application.
The overall obfuscation score of the application is lastly computed as a weighted sum of the three techniques scores in the same way of the single binary case.

Chapter 5

Client Application

The client of Code Guardian is what makes its sophisticated analysis capabilities accessible to end users. It has been designed to be available on all the major platforms including desktop, web, Android and iOS. This has been achieved thanks to the incredible versatility of **`Kotlin Multiplatform`** and **`Compose Multiplatform`** (**CMP**) which permit to not having to rely on platform-specific technologies or hybrid approaches like `WebViews` in order to reach multiplatform compatibility. Instead, KMP and CMP allow to share almost the entire codebase across all the target platforms while still being able to compile down to native binaries for each of them. This results in a high performance application that feels native on every platform and that is very easily maintainable due to the shared codebase.
While KMP is the engine of the business logic for all the platforms, CMP is the framework used to build the GUI. CMP uses a declarative programming paradigm like its Android only counterpart, Jetpack Compose. There would have been also the possibility to decouple the GUI of iOS from CMP and use `SwiftUI` instead in order to have a more native feeling on Iphone devices but this approach has been discarded in favor of the uniformity provided by CMP.
KMP and CMP are relatively new technologies and even if they are already quite ripe for production use, they naturally retain some limitations. For example, the web target based on `Kotlin/Wasm`, chosen in Code Guardian for its performance advantages over the `Kotlin/JS` target, is still in beta and so it presents some incompleteness.
Furthermore, CMP is where the limitations are more evident because some `Composable` components, which are the building blocks of the GUI in CMP, are not yet supported on all the platforms. This comes from the fact that CMP inherits its Composables from `Jetpack Compose`, which is limited to Android and so uses some Android specific APIs that may not be available or emulable on the other platforms.

5.1 Landing page and Basic Info Page

Code Guardian's first screen is its landing page, shown in Figure 5.1, which provides at first glance the list of the performed and ongoing analyses with their status and some little information about the package. The list of the analyses is filterable, orderable and paged on server side as all the other lists that will be presented later on in this chapter. From this page, it is possible to navigate to one of the analysis specific details or submit a new analysis request.

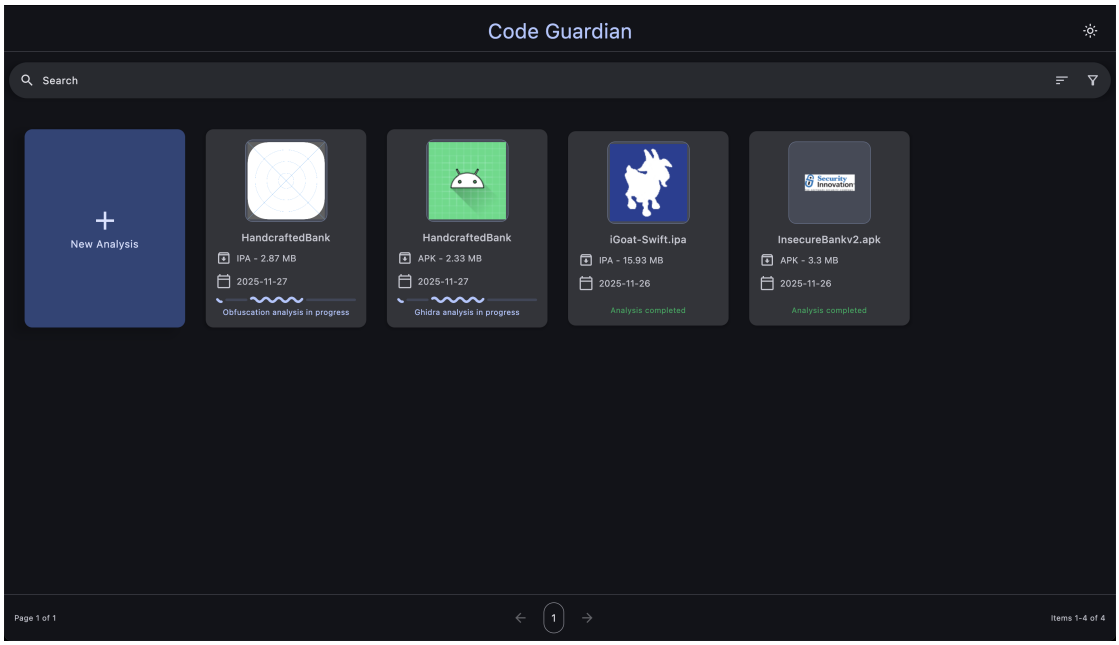

Figure 5.1: Landing Page

When an analysis is selected, the user navigates to the basic info page of the analysis, shown in Figure 5.2 in its desktop and mobile versions. This page shows the general information about the analysed package such as its name, size, platform and SDK version. This view also provides an overview of the categories of permissions requested by the application, with the maximum risk level among them and the locales supported by the application.

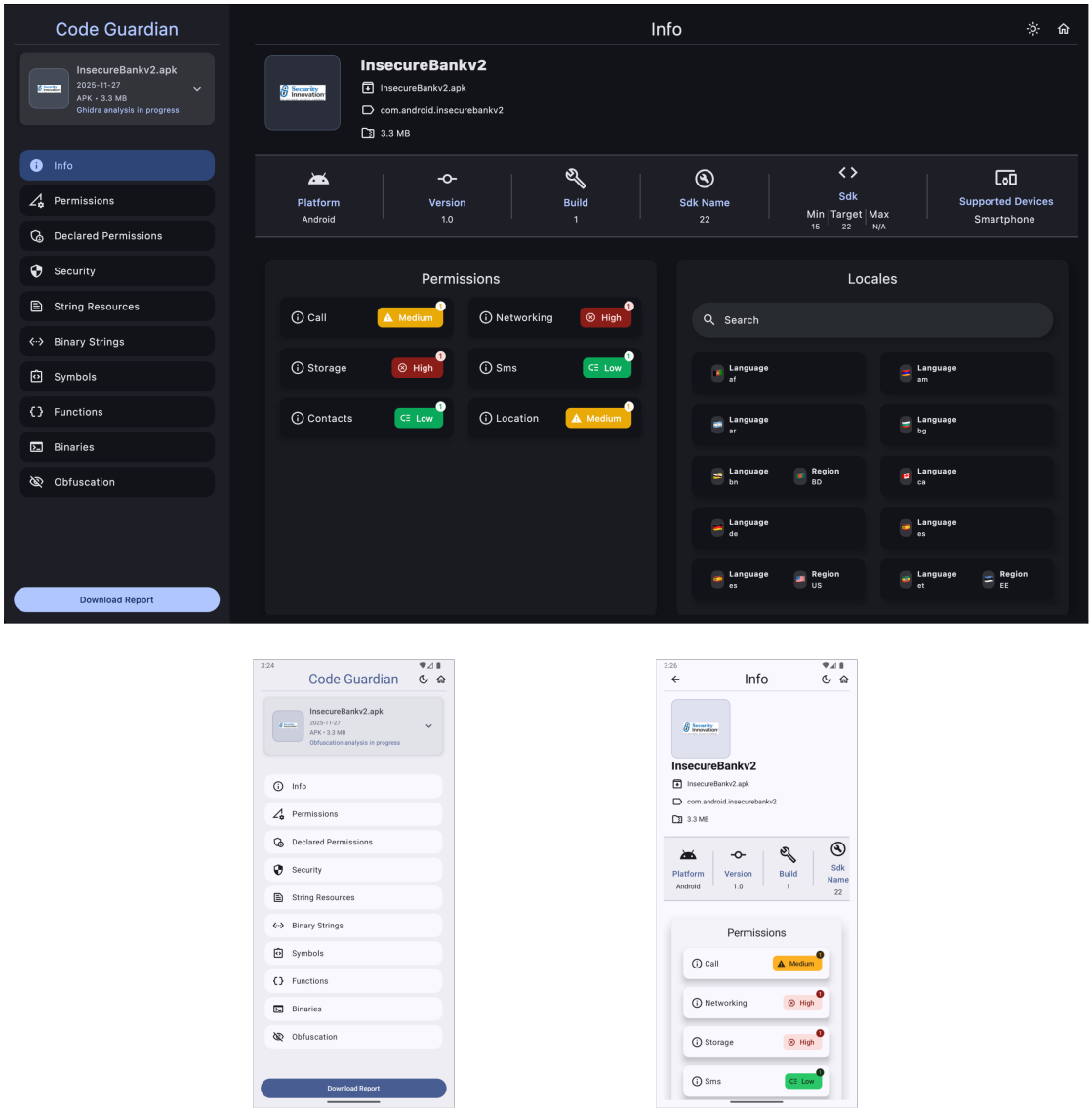

Figure 5.2: Basic Info Page

In this page it is visible the general structure of the client appplication which relies on a list and detail pattern: the left side contains the sidebar with the list of the sections and the right contains the details of the selected section. This pattern is implemented with a custom porting of the `NavigableListDetailScaffold` Composable of Jetpack Compose adapted to CMP. On mobile devices, the sidebar and details are shown as separate screens while on desktop and web they are visible side by side.

5.2 Obfuscation and Binaries Pages

The obfuscation evaluation page is where all the results and scores computed by the server during the analysis are shown. This page, shown in Figure 5.3, is composed by the scores related to the whole package and a list of the binaries present in it. Each binary card can then be clicked to navigate to the binary specific obfuscation scores and extracted elements.
Every binary of the package has its own dedicated page where all its obfuscation scores, binary protections and extracted components are shown.

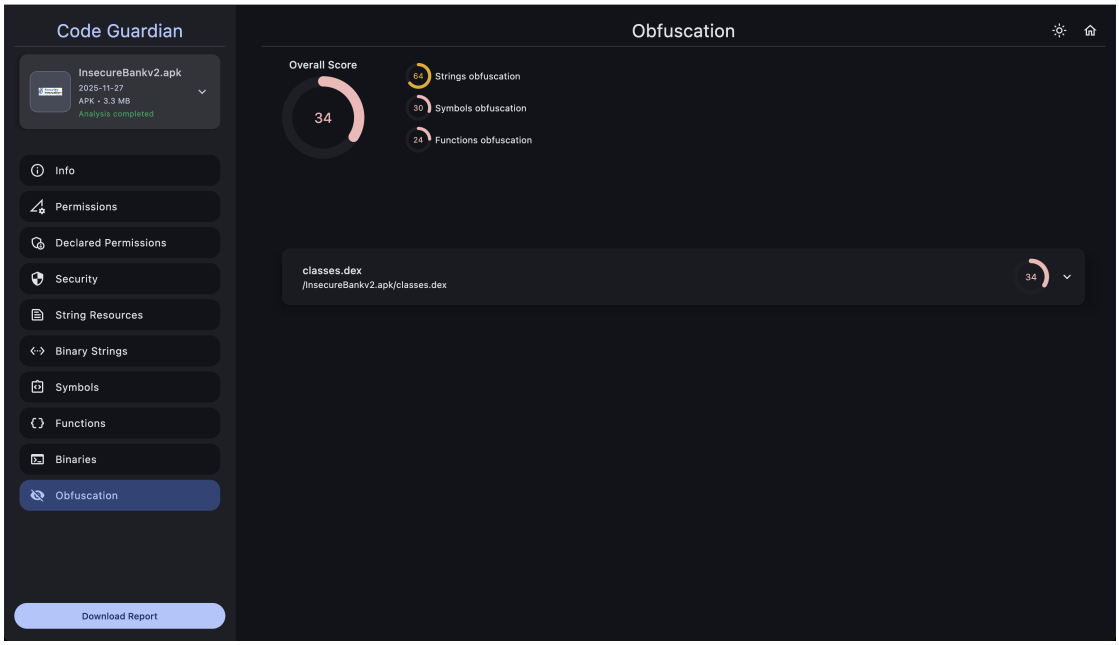

Figure 5.3: Obfuscation Page

5.3 Other pages

The other details sections of the client application are quite similar among them. They share the same list structure to show the elements that they are responsible for in cards, the binary strings page is reported in Figure 5.4 as an example.

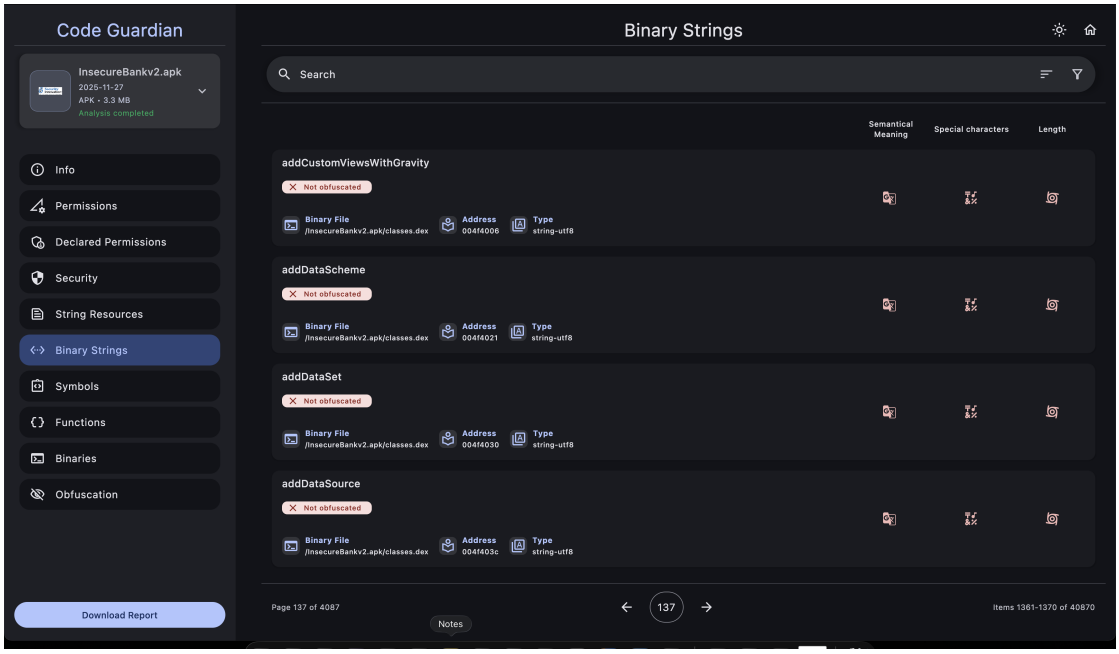

Figure 5.4: Binary Strings Page

All of these pages provide filtering, ordering and pagination features which are obviously different depending on the type of elements that are shown. For instance, the binary strings page allows to filter out the strings that are meaningful or that contain special characters while the security finds page allows to filter its elements by their risk level. The filters are implemented as a chips bar right below the search bar, each chip allows to open the corresponding filter drawer or just toggles the filter.
The cards representing the elements are designed to follow the same style across the pages even if each of them is adjusted to show the specific information related to the element. THe ones of security finds, permissions and string resources are shown in the following Figure 5.5

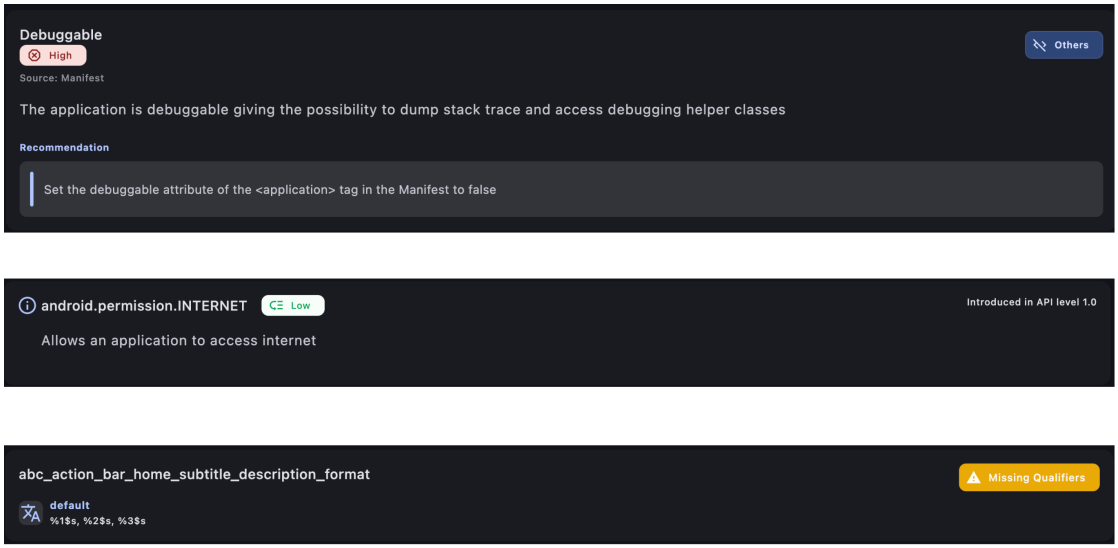

Figure 5.5: Cards of Code Guardian Elements

5.4 View Models Management

The `View Model` layer of the client application is where the client's business logic resides. In `KMP` and `CMP`, `View Models` are shared across all the platforms with the same strategy used in Android development with `Jetpack Compose`. In Code Guardian, they are implemented using coroutines and managing `StateFlows` to provide the needed data to the `Composables`. The states are mostly implemented as sealed interfaces to be able to clearly distinguish between loading, error and success states.
The data needed by the `Composables` of each page is not always immediately ready. For example, when an analysis is ongoing, fetching the binary strings would require waiting for the binary analysis to be completed first. Recreating the `View Models` each time the user navigates to the corresponding page would result in resource waste and bad user experience because any data already available and fetched would be lost. This aspect is addressed by the `@KoinViewModel` annotation and the `koinViewModel` function.
The annotation manages the dependency injection of the `ViewModel` but most importantly the method allows to control their lifecycle properly. In particular, by default this strategy scopes the `View Model` to the navigation graph so that it gets created once when the user first navigates to the corresponding page and it is retained in memory until it exits the graph. Furthermore, the `View Models` need to be refreshed when the observed analysis changes but also when the data needed

by them becomes suddenly available. This is achieved by using the `key` parameter of the `koinViewModel`, as shown in Listing 5.1, which forces the recreation of the `View Model` each time the key changes. The key contains the ID of the analysis currently observed and one or more boolean flags that indicate if the needed step of the analysis has been completed or not, so that when the step is completed the `View Model` gets recreated and fetches the newly available data.

BasicInfoPage.kt

```kotlin
val viewModel: BasicInfoPageViewModel = koinViewModel(
    key = "${analysis.id}_${
        isStepCompleted(
            requiredStatus = AnalysisStatus.BASICINFO,
            currentStatus = analysis.status
        )
    }"
)
```

Listing 5.1: Example of `koinViewModel()` with `key` parameter

5.5 Report Generation

The report generated by Code Guardian is the final output of the whole analysis process and it is thought as the primary system to share the analysis results outside of the client application itself. The report is generated in `PDF` format and it has a predefined structure that is the same for all the platforms. As shown in Figure 5.6, it contains all the basic information about the analysed package, all the security findings and the obfuscation analysis results.

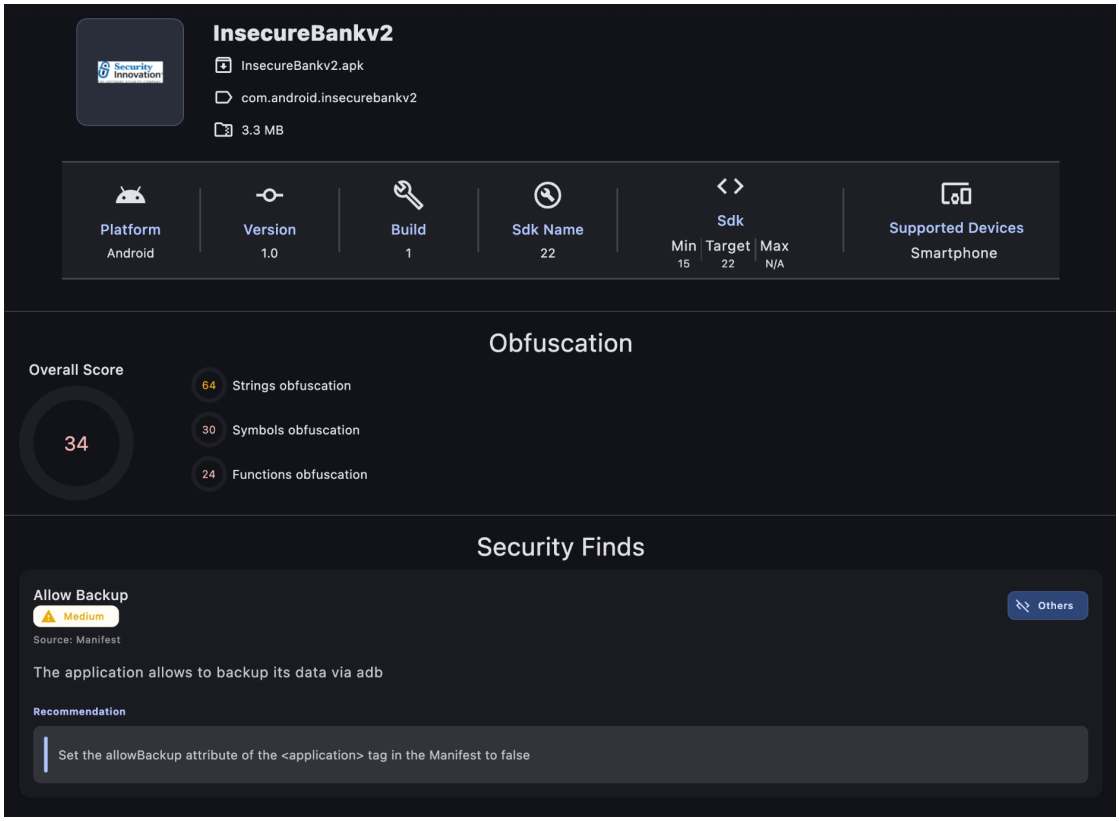

Figure 5.6: Code Guardian Report Example

At the beginning, the goal was to generate the report on server side after the analysis was completed and then serve it to the client as a static file. However, this strategy has been discarded because the report appearance was intended to be similar to the client application UI and so generating it completely on server side would have required to replicate the whole UI in an HTML template or similar, which would have been difficult to develop and above all to maintain. Even a simple modification to a component of the client UI which is part of the report, would have required a corresponding modification of the server template. This could result in too tight coupling between the client and the server creating a maintenance nightmare.
For this reason, the report generation task has been split in two parts: the client, under the hood, renders a `Composable` that represents the report asking the server only for the data that it needs to fill it. Then, once all the data has been retrieved from the server, the client exploits the capturable nature of Composables to snapshot it as an image and send this image to the server. The image is inherently compliant with the UI style of the application since it is generated by a Composable built with the same design. In this phase, the client takes care also of the report pagination

by keeping track of the height of the rendered content and intentionally inserting blank spaces within the `Composable` when a page break is needed, without cutting off any content. The server finally receives the image and, using `OpenPDF` library, builds the `PDF` report by just embedding the image inside it and paginating it as a normal A4 `PDF` document.
The downside of this strategy is that the report is just an image embedded in a `PDF` so it is not possible to select or copy text but this limitation has been accepted in favor of the maintainability that it provides.

Chapter 6

Testing, Future Developments and Conclusion

6.1 Testing

The analysis implemented in Code Guardian has been tested with a set of packages containing known vulnerabilities or weaknesses in order to verify the real effectiveness and reliability of the tool. The packages used for the tests have been selected from the pool of applications suggested by the OWASP MASTG. Some additional packages have been handcrafted in order to test specific scenarios which were not easily verifiable with the applications suggested by OWASP MASTG.
The tests have been performed by just running the analysis on the selected packages and then checking if the security finds reported by Code Guardian were actually present in the packages.
In particular, the applications used for the tests are *InsecureBankv2*[16], *iGoat-Swift*[17] and some of the `APK` packages provided by OWASP MASTG demo tests.
The verifications for misconfigurations illustrated in Section 4.3 have been successfully tested, for example the tool detected the allowed backup in *InsecureBankv2* and the active debug in its non production build variant.
The strings analysis and obfuscation evaluation have been tested using some handcrafted packages purposely containing hardcoded credit cards, fake API keys and functions written in an intentionally intricate and obfuscated way. In this case, Code Guardian was able to identify the flattened functions, obfuscated symbols and hardcoded secrets with some minor exceptions. As it was predicted, predicates that are opaque because their condition is logically always true or false but not

calculated on constants have not been detected at all.
The LLM proved to be effective but not always usable because its filters sometimes blocked certain input strings that were considered as violating the policies of the model itself like the strings referring to jailbreaking. To go beyond this limitation a solution could be to use maybe a local LLM instance without filters.
The tests to verify the obfuscation evaluation have been performed by modifying the content of the handcrafted packages and observing how the scores changed. The packages were little so the scores were really sensitive also to small changes because they had a small number of strings, symbols and functions but they were still coherent with the modifications.

6.2 Future Developments

Code Guardian presents itself as a ready to use tool for the analysis of mobile applications but it has huge room for future improvements on several things like performance, reliability and analysis features.

6.2.1 Performance Improvements

In terms of performance, the analysis is really heavy and time demanding, especially when dealing with large applications. This happens mainly because of the computational requirements of `Ghidra` when performing disassembling and above all decompiling. Furthermore, the LLM interaction needed for the evaluation of the semantical meaning of strings and symbols is also quite time consuming because binaries can contain a huge amount of strings and symbols to be evaluated, leading to a large number of requests to the LLM that slow down the analysis process. This second aspect can be improved by implementing some context caching mechanism to avoid sending the fixed parts of the prompt that explain the context multiple times, reducing the transmitted data and thus the time and cost of this task.
However, `Ghidra` represents the real and critical bottleneck of the analysis, relying on a single container running `Ghidra` for every analysis is furtherly improvable. Currently, the analyses are constrained to run sequentially waiting for the previous one to end because of the single `Ghidra` instance. Additionally, it is not possible to speed up a single analysis by using multiple `Ghidra` instances in parallel so a good approach could be to give to the server the possibility to spawn multiple containers running `Ghidra`, one for each submitted analysis, in order to perform the `Ghidra` related tasks in parallel for different analyses. This horizontal scaling would lead to a significant reduction of the waiting time for an incoming analysis request when the server is already busy processing another one.

6.2.2 Reliability Improvements

The server has been developed as a single node, resulting in having a single point of failure. If the server goes down for any reason, the ongoing analyses will be interrupted and the whole service will be unavailable until it gets back online. To overcome this limitation, a possible approach could be to horizontally scale also the server itself by deploying multiple instances managed by a load balancer that would be in charge of distributing the analyses requests among the available instances. Thus, moving from a monolithic to a distributed architecture would obviously improve reliability but it would simultaneously introduce consistency challenges, since the server nodes would need to be synchronized in order to share not only the information about the analyses but also the track of all the clients that have requested updates about certain analyses. This tracking is currently implemented as a simple data structure because of the single node architecture but it would need to be replaced with some appropriate distributed approach such as `Redis`.

6.2.3 Analysis Features Improvements

The improvement that comes to mind first when thinking about future developments of Code Guardian is indeed, without any doubt, the support for dynamic analysis. As things are standing now, Code Guardian can perform static analysis only, which is powerful but still limited because it would be blind in front of certain security aspects that can be checked only at runtime. The dynamic analysis could be implemented by integrating existing and reliable tools like `Frida`[20], which is also suggested by OWASP MASTG, in order to hook into the application at runtime and monitor its behaviour. A drawback of the dynamic analysis is that it would require the app to be executed in some way. Android apps could be run easily on the emulator provided by `Android Studio`, while iOS apps would need to be run on a physical device because the iOS simulator that comes with `Xcode`, unlike the Android emulator, does not emulate the hardware architecture of the device and so its capabilities are limited. Moreover, dynamic analysis can be automated only to a certain extent because often the context of the application needs to be understood in order to interact with it in the right way and every application has its own logic and flows.
Also the static analysis itself could be furtherly improved by integrating more security inspections like for example checking for other binary protections like `RELRO` and `ARC` or adding more `Semgrep` rules to scan better the decompiled code. Lastly, in terms of resources analysis, Code Guardian could be improved by implementing a system to detect the presence of incoherent and superfluous files in the package such as images not related to the banking domain as well as audios or videos which should not be present in a banking application. The resource analysis could also be furtherly enriched by scanning the formats of files contained in the

package reporting if some their format is improvable to save some space like for example using `WebP` instead of `JPEG` for images.

6.3 Conclusion

To conclude, Code Guardian has been designed and developed to be a tool to assist the vulnerability assessment process of mobile banking applications. It is capable of providing a robust and powerful obfuscation evaluation, several security inspections and a shareable report to summarize the results. All of this, accessible by the major platforms and integrating a large range of technologies and tools such as `Ghidra` and a LLM in a seamless way.
It represents a solid starting point for the vulnerability assessment and it has a huge potential to be furtherly enriched with new features and improvements thanks to its linear analysis workflow and modular integration of the different employed tools. Additionally, it is built as a tool oriented to be used on banking applications but its analysis capabilities can be applied to every type of mobile application, making it really versatile and useful in several contexts.

Bibliography

[1] European Union Agency for Cybersecurity (ENISA). *ENISA Threat Landscape: Finance Sector*. 2025. URL: https://www.enisa.europa.eu/sites/default/files/2025-02/Finance%20TL%202024_Final.pdf (cit. on p. 1).

[2] ThreatFabric Research Team. *Anatsa Trojan Returns: Targeting Europe and Expanding Its Reach*. 2024. URL: https://www.threatfabric.com/blogs/anatsa-trojan-returns-targeting-europe-and-expanding-its-reach (cit. on p. 1).

[3] Agenzia per la Cybersicurezza Nazionale (ACN). *ToxicPanda: Rilevata diffusione in Italia del nuovo trojan bancario*. 2024. URL: https://www.acn.gov.it/portale/w/toxicpanda-rilevata-diffusione-in-italia-del-nuovo-trojan-bancario-al03/241106/csirt-ita- (cit. on p. 2).

[4] ThreatFabric Research Team. *Brokewell: do not go broke from new banking malware!* 2024. URL: https://www.threatfabric.com/blogs/brokewell-do-not-go-broke-by-new-banking-malware (cit. on p. 2).

[5] ThreatFabric Research Team. *The Rise of RatOn: From NFC Heists to Remote Control and ATS*. 2025. URL: https://www.threatfabric.com/blogs/the-rise-of-raton-from-nfc-heists-to-remote-control-and-ats (cit. on p. 2).

[6] OWASP. *OWASP Mobile Application Security*. OWASP MAS project. 2025. URL: https://mas.owasp.org (cit. on pp. 2, 3).

[7] OWASP. *OWASP Mobile Top 10 Risks*. 2024. URL: https://owasp.org/www-project-mobile-top-10/ (cit. on p. 2).

[8] Android. *Android Developers Documentation*. Official Android developer documentation. 2025. URL: https://developer.android.com (cit. on p. 3).

[9] Apple. *Apple Platform Security*. Apple Platform Security guide. 2024. URL: https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf (cit. on p. 3).

[10] NIST. *NIST*. URL: https://www.nist.gov (cit. on p. 4).

[11] *apktool*. URL: https://apktool.org (cit. on p. 22).

[12] *gitleaks*. URL: https://github.com/gitleaks/gitleaks (cit. on p. 22).

[13] *radare2*. URL: https://rada.re/n/radare2.html (cit. on p. 23).

[14] National Security Agency (NSA). *Ghidra*. URL: https://github.com/NationalSecurityAgency/ghidra (cit. on p. 23).

[15] *Semgrep*. URL: https://semgrep.dev (cit. on p. 23).

[16] *InsecureBankv2 APK*. URL: https://github.com/dineshshetty/Android-InsecureBankv2 (cit. on pp. 31, 66).

[17] *iGoat-Swift IPA*. URL: https://github.com/OWASP/iGoat-Swift (cit. on pp. 35, 66).

[18] *Elf Header*. URL: https://refspecs.linuxfoundation.org/elf/gabi4+/ch4.eheader.html (cit. on p. 45).

[19] Apple OSS Distributions. *xnu*. URL: https://github.com/apple-oss-distributions/xnu (cit. on p. 46).

[20] *Frida*. URL: https://frida.re (cit. on p. 68).