UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

TFG del Grado en Ingeniería Informática

título del TFG
Documentación Técnica

Presentado por nombre alumno
en Universidad de Burgos — 6 de julio de 2025
Tutor: nombre tutor

Índice general

Índice de figuras

Índice de tablas

Apéndice A

Plan de Proyecto Software

A.1. Introducción

Este apéndice detalla los aspectos de gestión y planificación que han guiado el desarrollo de este Trabajo de Fin de Grado. Se presentará la planificación temporal seguida del estudio de viabilidad económica que estima los costes asociados al proyecto y, finalmente, un análisis de la viabilidad legal en base a las licencias del *software* empleado para determinar la licencia más adecuada para el producto final.

A.2. Planificación temporal

Para la gestión de este proyecto se ha adoptado un marco de trabajo inspirado en las metodologías ágiles, concretamente en SCRUM. La filosofía ágil, articulada en el Manifiesto Ágil [5], aboga por la colaboración, la flexibilidad ante los cambios y la entrega de valor de forma incremental, en contraposición a los modelos de desarrollo en cascada tradicionales. Aunque SCRUM está diseñado principalmente para la gestión de equipos, sus principios de organización en iteraciones o *sprints*, la revisión continua y la adaptación han sido muy útiles para estructurar el trabajo individual. En este caso, se ha seguido una versión «ligera» de SCRUM, con *sprints* de cuatro semanas, marcados por reuniones periódicas con el tutor para revisar los avances, resolver dudas y planificar los siguientes pasos. Para la gestión del tiempo diario, se han empleado técnicas de productividad como el método Pomodoro (usando aplicaciones como Forest [1]), alternando

bloques de trabajo intenso con descansos para mantener la concentración y evitar la fatiga.

A continuación, se detalla el cronograma del proyecto, dividido en los diferentes *sprints* realizados:

Sprint 1 (2 semanas: 2 diciembre - 15 diciembre)

Este fue el *sprint* inicial y más corto del proyecto, diseñado para asentar las bases y explorar las posibles líneas de investigación. El objetivo era encontrar un tema que combinara tanto el campo de la ciberseguridad como el de la inteligencia artificial. La primera idea que se exploró fue la de utilizar IA para mitigar vulnerabilidades de *hardware*, concretamente, ataques de ejecución especulativa (Spectre y Meltdown). Sin embargo, tras investigar acerca de ello en la literatura, se concluyó que era un campo extremadamente complejo y con muy poca documentación accesible, por lo que se descartó por su alta incertidumbre.

La investigación continuó pues hacia el análisis de *malware*. Como primer concepto, se consideró la idea de crear un analizador dinámico, una especie de *sandbox* donde ejecutar aplicaciones de forma aislada para estudiar su comportamiento en tiempo real. Aunque este enfoque contaba con más documentación, se determinó que la complejidad de desarrollar un entorno de este tipo desde cero era demasiado ambiciosa para el alcance del trabajo. Este *sprint* fue el más breve y acabó antes de tiempo debido, principalmente a su proximidad con los exámenes finales y a las vacaciones de Navidad.

Sprint 2 (4 semanas: 20 enero - 16 febrero)

Tras las vacaciones, se retomó la investigación centrándose en el análisis de *malware*, pero esta vez explorando otras técnicas como el análisis estático y el híbrido. El análisis estático destacó inmediatamente por su principal ventaja: la capacidad de detectar *malware* sin ejecutarlo. La literatura confirmó que la combinación de análisis estático y modelos de aprendizaje profundo *deep learning* era un campo de investigación muy activo y con resultados prometedores.

Con esta idea clara, se procedió a elegir la plataforma. Se descartaron los ejecutables de Windows (PE) y Linux (ELF) por la dificultad de analizar código máquina compilado. La elección final fue el formato APK de Android, ya que su estructura, similar a un archivo comprimido, y su código Dalvik, fácilmente descompilable, simplificaban enormemente la extracción de características. Durante este *sprint* se encontraron los *papers* fundamentales

que servirían de guía para el resto del proyecto, especialmente el trabajo de İbrahim et al. [7], que se convirtió en la principal referencia de este.

Sprint 3 (4 semanas: 17 febrero - 16 marzo)

Con el rumbo del proyecto ya definido, este *sprint* se centró en comenzar la fase de prototipado. El primer paso fue buscar un *dataset* adecuado para las pruebas iniciales. Tras evaluar varias opciones, se seleccionó el *dataset* Drebin [4] por su gran tamaño y la similitud de sus características con las descritas en el *paper* de referencia. Se dedicó una parte importante del tiempo a desarrollar los *scripts* necesarios para procesar el formato particular de Drebin y convertirlo en un archivo CSV manejable. Paralelamente, se comenzó a redactar la memoria del proyecto, documentando los primeros conceptos teóricos de ciberseguridad, y se integró la herramienta Poetry para facilitar con la gestión de dependencias.

Sprint 4 (4 semanas: 17 marzo - 13 abril)

Este *sprint* fue de carácter principalmente teórico y de diseño. A medida que avanzaba la investigación sobre cómo construir una red neuronal desde cero, se fue ampliando la sección de conceptos de la memoria con las definiciones de IA, aprendizaje automático, preprocesamiento de datos, etc. Se estudió en profundidad el funcionamiento de las redes neuronales, las funciones de pérdida, los optimizadores y las arquitecturas de *embedding*. El objetivo era adquirir toda la base teórica necesaria antes de empezar a escribir el código del modelo prototipo.

Sprint 5 (4 semanas: 14 abril - 11 mayo)

Durante este *sprint* se desarrolló la primera versión del modelo de red neuronal en PyTorch, basado en el *dataset* Drebin. Durante este proceso surgió el dilema del *embedder*: se tomó la decisión clave de separar la arquitectura en un *embedder* (preprocesador) y una cabeza clasificadora (MLP) para poder comparar el modelo con algoritmos clásicos de ML. Se implementó el proceso de entrenamiento para la red neuronal y para los modelos clásicos, se unificó el sistema de guardado y carga de modelos, y se generaron las primeras gráficas y estadísticas de rendimiento, que validaron la viabilidad del enfoque.

Sprint 6 (4 semanas: 12 mayo - 8 junio)

Con el prototipo validado, este *sprint* se dedicó a refinar el código y a preparar la transición hacia el modelo final. Se refactorizó la implementación actual del modelo para separarlo en componentes más modulares y fáciles de mantener y modificar. El proceso de entrenamiento se mejoró para incluir la estratificación de los datos y tener en cuenta el desbalance de clases. Al mismo tiempo, comenzó la creación del *dataset* propio, se realizaron pruebas con Androguard para la extracción de características y se descubrió el repositorio AndroZoo [3], con el que se experimentó para automatizar la descarga de APKs. Finalmente, se desarrolló el *pipeline* completo para la creación del nuevo *dataset* y se adaptó el código del modelo para que fuera compatible con el.

Sprint 7 (4 semanas: 9 junio - 6 julio)

Este fue el *sprint* final y más intenso. La adaptación del modelo al nuevo y mucho más complejo *dataset* reveló graves problemas de rendimiento y diseño que habían pasado desapercibidos. Se dedicó un gran esfuerzo a la depuración y optimización del modelo, solucionando problemas de gestión de memoria, cuellos de botella en el procesamiento de datos y la inhabilidad del modelo para entrenar. Una vez solucionados, se reentrenó el modelo final y se realizó el análisis de resultados comparativo y de interpretabilidad. Paralelamente, se desarrolló la aplicación web de demostración con Streamlit, se creó el repositorio de despliegue con Docker y se desplegó la aplicación en un servidor de la universidad. Finalmente, se terminó de redactar toda la documentación del proyecto; la memoria y sus anexos.

A.3. Estudio de viabilidad

En este apartado se realiza un análisis de la viabilidad del proyecto desde dos perspectivas: la económica, estimando los costes asociados a su desarrollo, y la legal, estudiando las licencias del software utilizado para determinar la licencia más apropiada para el trabajo resultante.

Viabilidad económica

A continuación, se calcularán los costes teóricos asociados al desarrollo de este proyecto, considerando un escenario profesional hipotético en el que se contratara a personal y se adquirieran los recursos necesarios. Se desglosarán los costes en personales, de hardware, de software e indirectos.

Costes de personal

Son los costes asociados al salario de las personas que han trabajado en el proyecto. Se estima una dedicación de unas 650 horas a lo largo de 6 meses, lo que equivale a unas 28 horas semanales. Tomando como referencia el salario medio de un ingeniero júnior en España (aproximadamente 1 930 € brutos/mes por 40 horas semanales), el salario proporcional sería:

$$\frac{28 \text{ h/semana}}{40 \text{ h/semana}} \times 1\,930 \text{ €/mes} = 1\,351 \text{ €/mes}$$

A este salario hay que sumarle las cotizaciones a la Seguridad Social a cargo de la empresa. Según las bases de cotización vigentes [2], los tipos aplicables para contingencias comunes y profesionales suman aproximadamente un 29.9 % (23.6 % por contingencias comunes, 5.5 % por desempleo, 0.2 % FOGASA y 0.6 % por formación profesional). Por tanto, el coste total del desarrollador para la empresa durante los 6 meses sería:

$$1\,351 \text{ €} \times (1 + 0.299) \times 6 \text{ meses} = 10\,529.71 \text{ €}$$

Adicionalmente, se estima el coste del tutor del proyecto, con una dedicación de 1 hora semanal (4 horas/mes) y una tarifa supuesta de 30 €/hora.

$$(30 \text{ €/h} \times 4 \text{ h/mes}) \times (1 + 0.299) \times 6 \text{ meses} = 935.28 \text{ €}$$

Por tanto, el coste total en personal es de 11 464.99 €.

Costes de *hardware* (materiales)

El proyecto se desarrolló en un ordenador personal valorado en 1 600 €. Suponiendo una vida útil de 8 años, el coste de amortización para los 6 meses de proyecto es:

$$\frac{1\,600 \text{ €}}{8 \text{ años}} \times \frac{6 \text{ meses}}{12 \text{ meses/año}} = 100 \text{ €}$$

Además, para el entrenamiento del modelo se podría haber utilizado un servicio de *cloud computing*. Una estimación para una instancia pequeña durante un supuesto de 150 horas de trabajo en Google Cloud Platform sería de aproximadamente 108.65 €. Por lo cual, el coste total de *hardware* sería de 208.65 €.

Costes de *software*

Todo el *software* empleado en el desarrollo del proyecto es de código abierto y gratuito, con la única posible excepción del sistema operativo. Asumiendo el uso de una licencia de Windows 11 Home, los costes por *software* serían de 145 €.

Costes indirectos

Estos costes incluyen los suministros necesarios para el desarrollo. Suponiendo un consumo eléctrico medio de 50 kWh/mes a un precio de 0.1702 €/kWh y un coste de conexión a *internet* de 30 €/mes, los costes indirectos para los 6 meses de proyecto serían de 231.06 €:

$$(50 \text{ kWh/mes} \times 0.1702 \text{ €/kWh} \times 6 \text{ meses}) +$$
$$(30 \text{ €/mes} \times 6 \text{ meses}) = 51.06 \text{ €} + 180 \text{ €} = 231.06 \text{ €}$$

Coste total del proyecto

La suma de todas los costes anteriores nos da una estimación del coste total teórico del proyecto.

Concepto	**Coste Estimado**
Costes de personal	11 464.99 €
Costes de hardware	208.65 €
Coste de software	145.00 €
Costes indirectos	231.06 €
Coste total	12 049.70 **€**

Tabla A.1: Resumen de los costes totales del proyecto.

Dado que el proyecto es de carácter puramente académico y de investigación, no se contempla la posibilidad de comercializarlo, por lo cual, no se calculan los posibles beneficios de este.

Viabilidad legal

Este apartado analiza las licencias del software utilizado para determinar bajo qué licencia puede ser distribuido este proyecto, garantizando el cumplimiento de todos los términos legales.

Tipos de licencias *Open Source*

Existen numerosas licencias de código abierto, cada una con diferentes permisos y obligaciones. La siguiente tabla resume algunas de las más comunes.

	MIT	BSD-2	BSD-3	Apache 2.0	LGPL-2.1	GPL-3.0
Permisos						
Uso comercial	x	x	x	x	x	x
Modificación	x	x	x	x	x	x
Distribución	x	x	x	x	x	x
Uso privado	x	x	x	x	x	x
Patentes concedidas				x		(implícitas)
Condiciones						
Conservar aviso/licencia	x	x	x	x	x	x
Indicar cambios				x	x	x
Publicar código derivado					(solo lib)	x
Misma licencia en derivado					(solo lib)	x
No uso del nombre/marca			x	x*		
Limitaciones						
Sin garantía	x	x	x	x	x	x
Sin responsabilidad	x	x	x	x	x	x

Tabla A.2: Características de licencias *open source*. *Solo marcas registradas.

Licencias del software empleado

La gran mayoría de las herramientas y librerías utilizadas en este proyecto se distribuyen bajo licencias de código abierto muy permisivas.

Elección de la licencia del proyecto

Como se muestra en la tabla A.3, la mayoría de las dependencias utilizan licencias permisivas como MIT, BSD y Apache 2.0, que permiten el uso, modificación y distribución del software con muy pocas restricciones. El único caso especial es Androguard, licenciado bajo LGPLv2.1. Esta licencia exige que si se modifica el código fuente de la librería, dichas modificaciones deben publicarse bajo la misma licencia. Sin embargo, como este proyecto utiliza Androguard como una librería externa, sin modificar su código, no estamos obligados a aplicar la licencia LGPL a nuestro propio código.

Dada la naturaleza permisiva de las licencias de las dependencias, la opción más adecuada para este proyecto es una licencia igualmente permisiva.

Software / Herramienta	**Licencia**
Python	PSF License (similar a BSD)
Poetry, Optuna, SHAP	MIT License
Conda, Jupyter, PyTorch, Scikit-learn, NumPy, Pandas, UMAP	BSD 3-Clause License (o similar)
Matplotlib	PSF/BSD-style License
Streamlit, Docker	Apache License 2.0
Androguard	GNU Lesser General Public License v2.1 (LGPLv2.1)

Tabla A.3: Licencias del software usado en el proyecto

Por tanto, se ha decidido licenciar este trabajo bajo la licencia MIT, ya que maximiza la libertad de uso y es compatible con el resto del ecosistema de herramientas.

Legalidad del análisis de aplicaciones

Un punto importante a considerar es la legalidad de desensamblar y analizar archivos APK. La legislación europea, concretamente la Directiva 2009/24/CE sobre la protección jurídica de los programas de ordenador [6], contempla excepciones al derecho exclusivo del autor. El artículo 6 permite la descompilación cuando sea indispensable para obtener la información necesaria para lograr la interoperabilidad de un programa creado de forma independiente. Aunque el fin de este proyecto es la seguridad, el principio es análogo: se analiza la aplicación para entender su funcionamiento e interoperabilidad con el sistema operativo con fines de investigación y defensa. Es crucial destacar que este proyecto no modifica ni redistribuye ninguna de las aplicaciones analizadas; únicamente extrae características de ellas para su estudio, una práctica ampliamente aceptada y considerada legitima.

Apéndice B

Especificación de Requisitos

B.1. Introducción

Este apéndice documenta la especificación de requisitos del trabajo desarrollado. Su propósito es definir de manera formal y detallada las capacidades, características y restricciones del sistema. Se establecen los objetivos generales del proyecto, se desglosa un catálogo de requisitos funcionales y no funcionales, y finalmente, se describen los principales casos de uso que ilustran la interacción del usuario con la aplicación.

B.2. Objetivos generales

Los objetivos generales representan las metas de alto nivel que se buscaron alcanzar con la realización de este trabajo, combinando tanto las aspiraciones de investigación como los entregables prácticos.

1. **Investigar el estado del arte:** Realizar un análisis de la literatura científica para comprender las técnicas actuales de detección de *malware* con IA y posicionar el proyecto en el panorama actual.

2. **Desarrollar un sistema de detección de extremo a extremo:** Construir un *pipeline* completo, desde la recolección de datos y la extracción de características hasta el entrenamiento y la evaluación de un modelo funcional.

3. **Alcanzar un alto rendimiento predictivo:** Lograr que los modelos desarrollados obtengan métricas de clasificación altas, con un enfoque especial maximizar el *recall* para minimizar los falsos negativos.
4. **Comparar diferentes arquitecturas de modelos:** Evaluar y contrastar el rendimiento de una red neuronal profunda frente a algoritmos de aprendizaje automático clásicos para obtener conclusiones sobre la eficacia de cada enfoque.
5. **Garantizar la interpretabilidad del sistema:** Implementar técnicas que permitan explicar las decisiones de los modelos, aportando transparencia y confianza a los resultados.
6. **Crear una aplicación de demostración:** Desarrollar una interfaz web interactiva que permita a un usuario probar y visualizar el funcionamiento de todo el sistema.

B.3. Catálogo de requisitos

A continuación se presenta el catálogo detallado de requisitos que el sistema debe satisfacer.

Requisitos funcionales

Los requisitos funcionales (RF) especifican lo que el sistema debe hacer. Describen las funcionalidades, tareas y servicios que la aplicación final debe proporcionar al usuario para cumplir con su propósito.

- **RF-1: Carga de archivos APK.** El sistema debe proporcionar una interfaz que permita al usuario seleccionar y subir un archivo con formato `.apk` para su posterior análisis.
- **RF-2: Análisis y clasificación de la aplicación.** Una vez subida una APK, el sistema debe ejecutar el *pipeline* de análisis completo. Esto incluye la extracción de características estáticas, el preprocesamiento de los datos a través del *embedder* y la ejecución de la inferencia con todos los modelos entrenados (la red neuronal y los clasificadores clásicos).
- **RF-3: Visualización de las predicciones.** La aplicación debe presentar al usuario los resultados de la clasificación de forma clara. Debe mostrar un veredicto general y una tabla detallada con la predicción

de cada modelo individual, incluyendo los porcentajes de confianza para las clases «benigno» y «malicioso».

  - **RF-3.1: Transparencia del proceso de extracción.** Para que el proceso no sea una «caja negra», la interfaz debe permitir al usuario inspeccionar las características «brutas» que han sido extraídas de la APK. Esto incluye las diferentes listas de permisos, actividades, servicios, receptores, así como propiedades del archivo como su tamaño o su *fuzzy hash*.
  - **RF-3.2: Visualización de datos procesados.** Además de los datos en «bruto», el sistema debe mostrar la representación numérica en la que se transforman. Esto implica visualizar tanto los datos *tokenizados* y escalados como el vector de *embeddings* final que se introduce en los clasificadores.

- **RF-4: Interpretabilidad de las predicciones.** La aplicación debe ofrecer explicaciones sobre las decisiones del modelo. Para ello, deberá generar y mostrar un conjunto de gráficos de SHAP, incluyendo la importancia global de las características y análisis locales que detallen qué factores han influido en la predicción de la muestra actual.

- **RF-5: Visualización del espacio de características.** El sistema debe ser capaz de generar una proyección 2D del espacio de *embeddings* mediante UMAP. En esta visualización, se debe mostrar la distribución de las muestras del *dataset* de fondo y resaltar la posición de la APK recién analizada, permitiendo al usuario entender su ubicación relativa respecto a las clases conocidas.

- **RF-6: Gestión del historial de análisis.** El sistema debe mantener un historial de las aplicaciones analizadas durante la sesión activa del usuario. La interfaz debe permitir al usuario seleccionar una entrada del historial para volver a cargar sus resultados y ofrecer una opción para borrar todo el historial de la sesión.

Requisitos no funcionales

Los requisitos no funcionales (RNF) describen los atributos de calidad y las restricciones bajo las cuales el sistema debe operar. No se refieren a qué hace el sistema, sino a «cómo» lo hace, definiendo aspectos como su rendimiento, fiabilidad o portabilidad.

- **RNF-1: Rendimiento y fiabilidad de los modelos.** Los distintos clasificadores deben alcanzar un alto nivel de rendimiento, definido por métricas de evaluación estándar. Específicamente, se establece como requisito clave obtener una métrica de *recall* cercana al 98 %, garantizando una detección muy alta de las muestras maliciosas.

- **RNF-2: Eficiencia del análisis.** El tiempo total desde que el usuario sube una APK hasta que recibe un resultado completo (incluyendo la extracción, el preprocesamiento y la inferencia) debe ser razonable para una buena experiencia de usuario, idealmente completándose en menos de un par de minutos.

- **RNF-3: Portabilidad y facilidad de despliegue.** Todo el sistema, incluyendo la aplicación web, el modelo y sus dependencias, debe estar empaquetado en un contenedor Docker. Esto asegura que la aplicación sea portable y pueda ser desplegada de forma sencilla y consistente en diferentes sistemas operativos.

- **RNF-4: Usabilidad de la interfaz.** La aplicación web debe tener una interfaz de usuario clara, intuitiva y fácil de navegar para un perfil de usuario con conocimientos técnicos, pero no necesariamente experto en inteligencia artificial. La información debe presentarse de forma organizada y comprensible.

- **RNF-5: Modularidad del código.** La base de código del proyecto debe seguir un diseño modular que separe claramente las distintas responsabilidades (extracción de datos, arquitectura del modelo, entrenamiento, etc.). Esto facilita el mantenimiento, la experimentación y el posible desarrollo futuro del sistema.

B.4. Casos de uso

Los casos de uso describen las interacciones entre un actor (en este caso, el «Usuario/Analista») y el sistema para alcanzar un objetivo. A continuación se presenta un diagrama esquemático (Figura B.1) y se detallan los tres casos de uso principales.

CU-1	**Analizar una nueva aplicación APK**
Versión	1.0
Autor	David Cezar Toderas
Requisitos asociados	RF-1, RF-2, RF-3
Descripción	El usuario sube un archivo APK al sistema para que este sea analizado por los modelos de IA y se muestren los resultados de la clasificación.
Precondición	El usuario tiene un archivo `.apk` válido y la aplicación web está en ejecución.
Acciones	1. El usuario accede a la aplicación web. 2. El usuario arrastra y suelta un archivo APK en la zona de carga o lo selecciona mediante el explorador de archivos. 3. El usuario presiona el botón «Analizar APK». 4. El sistema procesa el archivo: extrae las características, las pasa por el *embedder* y realiza la inferencia con todos los modelos. 5. La interfaz se actualiza para mostrar la pestaña de «Predicciones» con el veredicto de cada modelo. 6. El análisis se añade al historial de la sesión.
Postcondición	Se muestra una predicción de clasificación para la APK subida.
Excepciones	**E-1.1:** Si el archivo subido no es un APK válido, el sistema muestra un mensaje de error. **E-1.2:** Si el archivo supera el límite de tamaño, el sistema muestra una advertencia y no permite el análisis.
Importancia	Alta

Tabla B.1: CU-1 Analizar una nueva aplicación APK.

CU-2	**Interpretar el resultado de un análisis**
Versión	1.0
Autor	David Cezar Toderas
Requisitos asociados	RF-3.1, RF-3.2, RF-4, RF-5
Descripción	Tras analizar un APK, el usuario explora las diferentes pestañas de la interfaz para comprender en profundidad el resultado y el razonamiento del modelo.
Precondición	Se ha completado con éxito un análisis (CU-1).
Acciones	1. El usuario navega a la pestaña «Características Extraídas» para ver los datos brutos y procesados. 2. El usuario navega a la pestaña «Explicaciones» para visualizar los gráficos de importancia de características de SHAP. 3. El usuario navega a la pestaña «UMAP» para ver la proyección del *embedding* de la aplicación analizada.
Postcondición	El usuario obtiene una visión detallada de los datos y las justificaciones que respaldan la predicción del modelo.
Excepciones	Ninguna.
Importancia	Alta

Tabla B.2: CU-2 Interpretar el resultado de un análisis.

CU-3	**Gestionar el historial de análisis**
Versión	1.0
Autor	David Cezar Toderas
Requisitos asociados	RF-6
Descripción	El usuario interactúa con el historial de análisis de la sesión actual para consultar resultados anteriores o para limpiar la lista.
Precondición	Se ha analizado al menos una aplicación durante la sesión actual.
Acciones	**Flujo A: Consultar un análisis anterior** 1. El usuario hace clic sobre el nombre de un archivo en la lista del «Historial de Análisis». 2. El sistema carga en la vista principal todos los datos y resultados correspondientes a ese análisis. **Flujo B: Limpiar el historial** 1. El usuario hace clic en el botón «Limpiar Historial». 2. El sistema borra todas las entradas del historial de la sesión.
Postcondición	**Flujo A:** La interfaz muestra los resultados del análisis seleccionado. **Flujo B:** La lista del historial de análisis queda vacía.
Excepciones	Ninguna.
Importancia	Media

Tabla B.3: CU-3 Gestionar el historial de análisis.

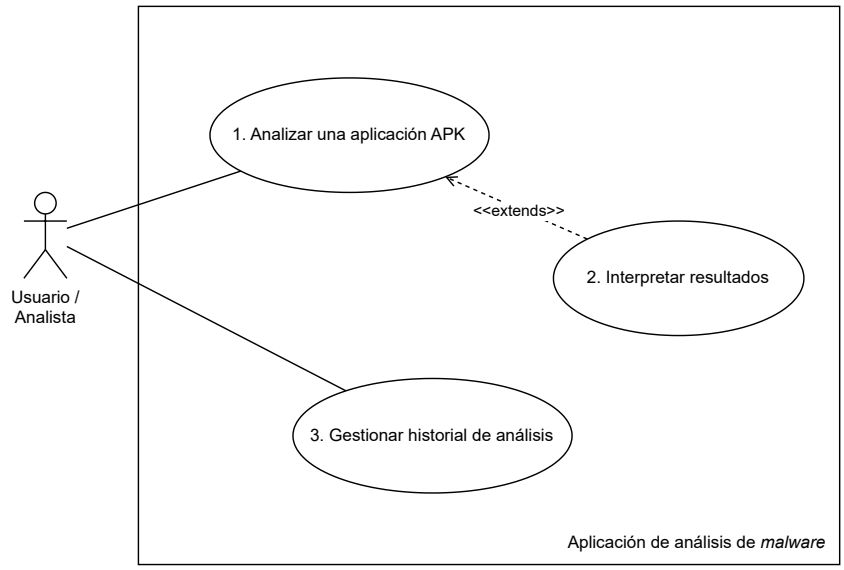


Figura B.1: Diagrama de casos de uso de la aplicación de análisis de *malware*.

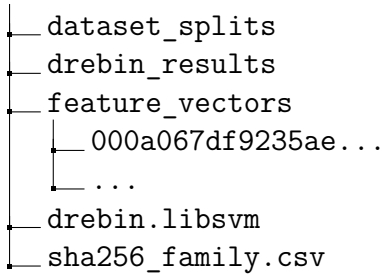
Apéndice C

Especificación de diseño

C.1. Introducción

C.2. Diseño de datos

```
/
├── assets
├── lib
│   ├── arm64-v8a
│   ├── x86
│   ├── x86_64
│   └── ...
├── META-INF
│   ├── CERT.SF (might have a different name)
│   ├── CERT.RSA (might have a different name)
│   └── MANIFEST.MF
├── kotlin
├── res
├── AndroidManifest.xml
├── classes.dex
└── resources.arsc
drebin
├── apks
│   ├── drebin_0.zip
│   ├── drebin_1.zip
│   └── ...
└── metadata
```



C.3. Diseño procedimental

C.4. Diseño arquitectónico

Apéndice D

C

Especificación de Diseño

D.1. Introducción

Este apéndice presenta el diseño técnico detallado del sistema desarrollado. Su objetivo es servir como un plano que describe la arquitectura del software, la estructura de los datos con los que opera y la lógica procedimental de sus operaciones clave. Se abordará el diseño desde tres perspectivas: el diseño de datos, que define la estructura de los archivos APK y los *datasets*; el diseño procedimental, que ilustra las interacciones principales mediante diagramas de secuencia; y el diseño arquitectónico, que desglosa la estructura interna del modelo de inteligencia artificial y de la aplicación web.

Este apéndice detalla el diseño técnico del sistema desarrollado, abarcando tres áreas fundamentales: el diseño de los datos, el diseño procedimental y el diseño arquitectónico. El objetivo es proporcionar una descripción clara y estructurada de cómo se ha organizado la información, cómo fluyen los procesos lógicos y cómo se ha construido la arquitectura del software, desde el modelo de inteligencia artificial hasta la aplicación web. Este documento sirve como un plano técnico que facilita la comprensión, el mantenimiento y la futura expansión del proyecto.

D.2. Diseño de datos

El diseño de datos es fundamental para cualquier sistema de aprendizaje automático. En esta sección se describe la estructura de los datos de entrada

primarios (archivos APK) y los conjuntos de datos estructurados que se han derivado de ellos para el entrenamiento y la evaluación de los modelos.

El diseño de datos es fundamental para cualquier sistema de aprendizaje automático. En esta sección se describe la estructura de los datos de entrada (archivos APK), el formato de los *datasets* utilizados durante el desarrollo y el *pipeline* de preprocesamiento que los transforma para que puedan ser consumidos por los modelos.

Estructura de un archivo APK

Un archivo APK (*Android Package Kit*) es el formato de paquete utilizado por el sistema operativo Android para la distribución e instalación de aplicaciones móviles. Aunque parece un único archivo, en realidad es un archivo comprimido (basado en el formato JAR y, por tanto, compatible con ZIP) que contiene un conjunto de ficheros y directorios con una estructura bien definida. La estructura de una APK es la siguiente:

```
/
├── assets
├── lib
│   ├── arm64-v8a
│   ├── x86
│   ├── x86_64
│   └── ...
├── META-INF
│   ├── CERT.SF
│   ├── CERT.RSA
│   └── MANIFEST.MF
├── kotlin
├── res
├── AndroidManifest.xml
├── classes.dex
└── resources.arsc
```

Los componentes más relevantes son:

- **`assets/`**: Contiene recursos brutos que la aplicación puede utilizar, como ficheros de configuración, bases de datos o recursos de *machine learning*.

- **`lib/`**: Como Android es multiplataforma, este directorio incluye el código compilado específico de las diferentes arquitecturas de procesador (ARM, `x86`). Contiene, a su vez, las librerías nativas de la aplicación.
- **`META-INF/`**: Almacena los metadatos de la firma de la aplicación. Los ficheros `CERT.SF` y `CERT.RSA` contienen el certificado y la firma que garantizan la autenticidad e integridad de la aplicación, mientras que `MANIFEST.MF` contiene los *hashes* de todos los ficheros del paquete.
- **kotlin:** Directorio que contiene el código fuente de Kotlin, si la aplicación está escrita en este lenguaje.
- **`res/`**: Contiene los recursos de la aplicación que no están compilados, como los diseños de la interfaz (*layouts*), las imágenes (*drawables*) o las cadenas de texto (*strings*).
- **`AndroidManifest.xml`:** Es el archivo más importante del APK. Es un fichero XML obligatorio que describe la información esencial sobre la aplicación al sistema Android, como su nombre, componentes (actividades, servicios), permisos, versiones SDK mínima y objetivo.
- **`classes.dex`:** Contiene el código de la aplicación compilado en formato DEX (*Dalvik Executable*), que es el que ejecuta la máquina virtual de Android (ART / Dalvik). Puede haber múltiples archivos .dex si la aplicación es grande.
- **`resources.arsc`:** Es un archivo que contiene recursos precompilados, como las cadenas de texto, para un acceso más eficiente por parte del sistema.

Diseño de los conjuntos de datos

A lo largo del proyecto se han utilizado dos *datasets* principales: el *dataset* público Drebin para la fase de prototipado y un *dataset* propio para el desarrollo del modelo final.

Dataset Drebin

El formato original del *dataset* Drebin no era una tabla, sino una estructura de directorios compleja. Tras un proceso de *parsing*, se transformó en un único archivo CSV con la siguiente estructura por fila:

```
sha256 | req_permissions | app_components | ... | malware
000a06 | ["INTERNET", ...] | [".GameService", ...] | ... | 1
```

Cada fila representa una aplicación, identificada por su *hash* SHA256, y cada columna contiene una lista de las características estáticas extraídas de la misma.

Dataset propio

El *dataset* final se construyó desde cero extrayendo características de 20.000 APKs obtenidas de AndroZoo. El formato resultante es un único fichero CSV donde cada fila corresponde a una aplicación. La estructura es la siguiente:

```
file_size | fuzzy_hash     | activities_list   | ... | opcode_counts
34234     | "ab34...cd56"  | ["com.app...", ...] | ... | [221, 3455, 4567,
```

Pipeline de preprocesado de datos

Para que los datos de los *datasets* puedan ser utilizados por los modelos, necesitan pasar primero por un *pipeline* de preprocesamiento dividido en dos fases:

1. **Procesamiento externo (offline):** Antes de iniciar cualquier entrenamiento, se realizan una serie de pasos sobre el *dataset* completo. Para cada característica categórica (como los permisos o las actividades), se construye un **vocabulario** que asigna un índice numérico único a cada posible valor. Luego, todas las listas de cadenas de texto se convierten en listas de índices. Finalmente, se aplica un **padding** para que todas las listas de una misma columna tengan la misma longitud, convirtiéndolas en matrices numéricas.
2. **Procesamiento interno (online):** Una vez los datos están en formato de matrices de índices, el *embedder* del modelo se encarga del resto. Durante el entrenamiento o la inferencia, aplica las capas de *embedding* para convertir estos índices en vectores densos. Además, las características numéricas escalares (como el `file_size`) son normalizadas internamente por el modelo para que sus rangos de valores no desestabilicen el entrenamiento.

D.3. Diseño procedimental

Para ilustrar el flujo de trabajo y las interacciones dentro del sistema, a continuación se presentan los diagramas de secuencia de los dos procesos

más importantes: el análisis de un APK por parte de un usuario y el proceso interno de entrenamiento de un modelo.

Diagrama de secuencia: Análisis de un nuevo archivo APK

Este diagrama muestra la secuencia de interacciones desde que un usuario sube un archivo a la aplicación web hasta que recibe una predicción.

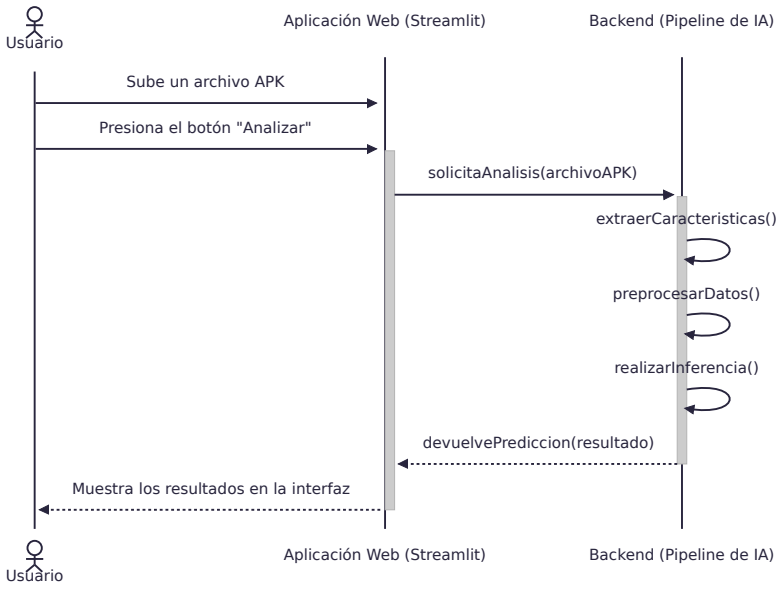


Figura D.1: Diagrama de secuencia del proceso de analizar una APK.

Diagrama de secuencia: Proceso de entrenamiento del modelo

Este diagrama detalla el bucle de entrenamiento de la red neuronal, un proceso iniciado por el desarrollador para ajustar los pesos del modelo.

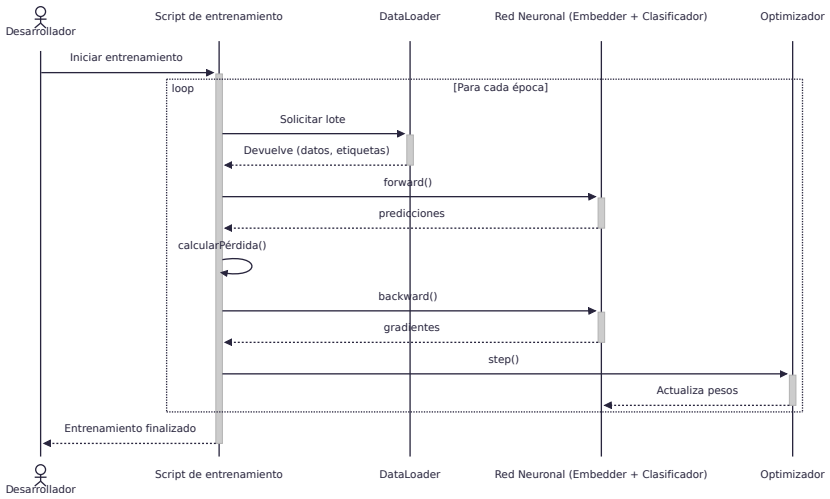


Figura D.2: Diagrama de secuencia del proceso de entrenamiento del modelo de red neuronal.

D.4. Diseño arquitectónico

En esta sección se detalla la arquitectura tanto del modelo de inteligencia artificial como de la aplicación web de demostración.

Arquitectura del modelo

En esta sección se detalla la arquitectura del software, tanto del modelo de inteligencia artificial como de la aplicación web que lo utiliza.

Arquitectura del modelo

El modelo de inteligencia artificial, implementado en PyTorch, sigue una arquitectura modular y flexible, compuesta por dos grandes bloques: el *embedder* y el clasificador.

```
APKAnalysisModel(
  (embedder): APKFeatureEmbedder(
  (seq_embedders): ModuleDict(...)
  (char_embedders): ModuleDict(...)
  (char_gru): ModuleDict(...)
  (vector_reducers): ModuleDict(...)
```

```
  )
  (classifier): APKClassifier(
   (mlp): Sequential(...)
  )
 )
```

- **APKFeatureEmbedder (Embedder):** Es el componente más complejo y el corazón del sistema. Su única responsabilidad es recibir las características preprocesadas de un APK y convertirlas en un único vector numérico denso. Para ello, contiene diferentes submódulos especializados:
  - **seq_embedders:** Un diccionario de capas 'Embedding' de PyTorch, una para cada característica de tipo lista (permisos, actividades, etc.). Cada capa aprende a representar los elementos de su vocabulario como un vector.
  - **char_embedders y char_gru:** Un módulo especializado para procesar el `FUZZY_HASH`. Trata el *hash* como una secuencia de caracteres, los convierte en vectores con un *embedding* y luego los procesa con una capa GRU (Gated Recurrent Unit) para capturar patrones secuenciales.
  - **vector_reducers:** Un diccionario de pequeñas redes neuronales (MLPs) que toman las características que ya son vectores numéricos (como `OPCODE_COUNTS`) y reducen su dimensionalidad para que sea consistente con la de los otros *embeddings*.

  La salida de todos estos submódulos, junto con las características escalares, se concatena para formar el vector final.
- **APKClassifier (Clasificador):** Este componente es la cabeza del modelo. Por defecto, es una red neuronal de tipo Perceptrón Multicapa (MLP) que toma el vector del *embedder* y, a través de una o más capas ocultas con funciones de activación ReLU y capas de *dropout*, lo procesa para obtener la predicción final en sus dos neuronas de salida.

Una de las claves de este diseño es que la cabeza clasificadora es **intercambiable**. La salida del *embedder* es un vector de características de alta calidad que puede ser utilizado para entrenar cualquier otro modelo de aprendizaje automático clásico (como RandomForest, XGBoost, SVM, etc.), tratando a la red neuronal simplemente como un potente paso de ingeniería de características.

Figura D.3: Diagrama de la arquitectura modelo de red neruonal.

Arquitectura de la aplicación web

La aplicación de demostración se ha desarrollado con Streamlit y sigue una arquitectura cliente-servidor simple.

- **Cliente:** Es el navegador web del usuario. Se encarga de renderizar la interfaz de usuario y de enviar las interacciones del usuario (como la subida de un archivo o el clic en un botón) al servidor.
- **Servidor:** Es el *script* de Python de Streamlit que se ejecuta en el servidor. Este se encarga de toda la lógica de la aplicación: recibe las peticiones del cliente, carga los modelos de IA de los artefactos guardados, ejecuta el *pipeline* de análisis sobre los datos recibidos y genera dinámicamente los componentes de la interfaz (tablas, gráficos, texto) que se envían de vuelta al cliente para ser mostrados.

Arquitectura de la aplicación web

La aplicación de demostración se ha desarrollado con Streamlit y sigue una arquitectura cliente-servidor simple.

- **Cliente (Navegador Web):** El usuario interactúa con la interfaz de la aplicación, que se renderiza en su navegador. Esta interfaz está compuesta por los componentes que ofrece Streamlit (botones, selectores, pestañas, etc.).
- **Servidor (Script de Python):** Un único *script* de Python se ejecuta en el servidor. Este *script* contiene toda la lógica de la aplicación. Se encarga de cargar los modelos entrenados en memoria al iniciarse, gestionar las interacciones del usuario (como la subida de archivos), llamar al *pipeline* de análisis para obtener las predicciones y generar los gráficos de interpretabilidad. Cuando el cliente realiza una acción, el servidor la procesa y le devuelve los nuevos elementos visuales para que la interfaz se actualice.

Este diseño permite un desarrollo muy rápido y una integración perfecta con todo el código de análisis y modelado desarrollado en Python.

Apéndice E

Documentación técnica de programación

E.1. Introducción

E.2. Estructura de directorios

E.3. Manual del programador

E.4. Compilación, instalación y ejecución del proyecto

E.5. Pruebas del sistema

Apéndice F

Documentación de usuario

F.1. Introducción

F.2. Requisitos de usuarios

F.3. Instalación

F.4. Manual del usuario

Apéndice G

Anexo de sostenibilización curricular

G.1. Introducción

Este anexo incluirá una reflexión personal del alumnado sobre los aspectos de la sostenibilidad que se abordan en el trabajo. Se pueden incluir tantas subsecciones como sean necesarias con la intención de explicar las competencias de sostenibilidad adquiridas durante el alumnado y aplicadas al Trabajo de Fin de Grado.

Más información en el documento de la CRUE https://www.crue.org/wp-content/uploads/2020/02/Directrices_Sosteniblidad_Crue2012.pdf.

Este anexo tendrá una extensión comprendida entre 600 y 800 palabras.

Bibliografía

[1] Forest - Stay focused, be present — forestapp.cc. https://www.forestapp.cc/. [Accessed 05-07-2025].

[2] Seguridad Social: Cotización / Recaudación de Trabajadores — seg-social.es. https://www.seg-social.es/wps/portal/wss/internet/Trabajadores/CotizacionRecaudacionTrabajadores/36537#36538.

[3] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[4] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[5] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.

[6] Unión Europea. Directiva 2009/24/CE del parlamento europeo y el consejo, de 23 de abril de 2009, sobre la protección jurídica de programas de ordenador. *Diario Oficial de la Unión Europea L*, 111(5):16–22, 2009.

[7] Mülhem İbrahim, Bayan Issa, and Muhammed Basheer Jasser. A method for automatic android malware detection based on static analysis and deep learning. *IEEE Access*, 10:117334–117352, 2022.