Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Universidad
Zaragoza

Trabajo de Fin de Máster
Máster Universitario en Ingeniería Informática

Clasificación de funciones en binarios ELF mediante *autoencoders* multicapa y análisis de programas

Autor:
Hong Christian Lin Jiang

Directores:
José Roldán Gómez
Ricardo Julio Rodríguez Fernández

Universidad de Zaragoza

Junio de 2025
Curso 2024/2025

Resumen

La detección y clasificación de programas dañinos sigue siendo uno de los principales desafíos abiertos en el ámbito de la ciberseguridad. Una de las líneas de investigación más prometedoras para abordar este problema consiste en el análisis de archivos binarios a bajo nivel, especialmente mediante la identificación automática de funciones y su clasificación según su procedencia o comportamiento. Poder distinguir si una función pertenece a una biblioteca estándar, a una implementación propia o a código potencialmente malicioso resulta clave en tareas de ingeniería inversa, análisis forense o detección de amenazas en sistemas ejecutables.

Este trabajo presenta un sistema para la clasificación de funciones extraídas de ejecutables de Linux, centrado en identificar su procedencia, como puede ser una biblioteca o paquete determinado. Se aborda el reto de trabajar con binarios compilados estáticamente y sin información simbólica, lo que impide utilizar métodos tradicionales basados en nombres o estructuras conocidas. Para ello, se propone una solución basada en la extracción de la representación del grafo de flujo de control, seguida de un modelo de aprendizaje automático que opera sobre dichos grafos.

El sistema desarrollado combina análisis estático con aprendizaje profundo sobre estructuras de grafos. Concretamente, dado un programa binario, se extraen sus funciones mediante la herramienta `radare2`, y se representan como grafos dirigidos. Posteriormente, estos grafos se analizan mediante bibliotecas de aprendizaje profundo y se utilizan para entrenar un modelo convolucional de grafos. La evaluación del sistema se realiza de manera cuantitativa a través de múltiples ejecuciones, con el objetivo de medir su precisión y robustez.

Los resultados experimentales obtenidos muestran que el sistema alcanza un rendimiento elevado y estable en tareas de clasificación entre bibliotecas, manteniendo incluso métricas competitivas en configuraciones más complejas. Estos resultados validan la viabilidad del enfoque propuesto, evidenciando que es posible identificar la procedencia de funciones de forma precisa mediante técnicas de aprendizaje profundo sobre representaciones basadas en grafos.

Abstract

The detection and classification of malicious software remains one of the major unsolved challenges in the field of cybersecurity. One promising line of research to address this issue involves the low-level analysis of binary programs, particularly through the automatic identification and classification of individual functions. Being able to distinguish whether a function belongs to a standard library, a custom implementation, or potentially harmful code is essential in reverse engineering, forensic analysis, and threat detection in executable systems.

This work presents a system for the classification of functions extracted from Linux executables, with a focus on identifying their origin, such as a specific library or software package. The main challenge lies in dealing with statically compiled binaries that lack symbolic information, which prevents the use of traditional methods based on function names or known structures. To overcome this limitation, the proposed approach extracts control-flow graphs for each function and applies a graph-based machine learning model for classification.

The system combines static analysis techniques with deep learning over graph structures. Specifically, functions are extracted from each binary using the tool `radare2` and represented as directed graphs. These graphs are then processed using graph-based deep learning libraries to train a convolutional graph neural network. The evaluation of the system is carried out quantitatively through multiple executions to assess its precision and robustness.

The experimental results show that the system achieves high and stable performance in function classification across libraries, maintaining competitive metrics even in more complex scenarios. These results validate the feasibility of the proposed approach and demonstrate that it is possible to accurately identify the origin of binary functions using deep learning techniques applied to graph-based representations.

Índice general

1. Introducción **1**
1.1. Contexto 1
1.2. Objetivo del proyecto 1
1.3. Estructura del documento 2

2. Fundamentos teóricos **3**
2.1. Formato ELF y análisis estático 3
2.2. Grafos de flujo de control 3
2.3. Aprendizaje profundo y *autoencoders* 4
2.4. Redes neuronales sobre grafos 5

3. Estado del arte **7**
3.1. Modelos para comparación de funciones binarias 7
3.2. Conjuntos de datos y herramientas para el entrenamiento 8

4. Arquitectura del sistema propuesto **11**
4.1. Descripción del sistema 11
4.2. Implementación técnica 13
4.3. Reproducibilidad del sistema 14
4.4. Limitaciones de la aproximación 14

5. Evaluación del sistema **15**
5.1. Metodología experimental 15
5.2. Métricas de evaluación utilizadas 15
5.3. Resultados experimentales 16
5.4. Discusión de resultados 18

6. Conclusiones **21**

Bibliografía **23**

A. Planificación temporal y dedicación horaria **25**

Capítulo 1

Introducción

1.1. Contexto

La detección y clasificación de programas dañinos continúa siendo uno de los grandes desafíos abiertos en el ámbito de la ciberseguridad. A pesar del avance en sistemas de defensa, muchas técnicas actuales siguen siendo frágiles frente a binarios ofuscados, desconocidos o compilados con herramientas no estándar. La necesidad de identificar patrones funcionales en código binario, incluso cuando no se dispone de su código fuente ni de información simbólica, ha motivado el desarrollo de enfoques más robustos basados en el análisis estructural y semántico del código [5, 22].

En la actualidad, el análisis de programas compilados ha adquirido una relevancia creciente en el ámbito de la ingeniería inversa, la ciberseguridad y la auditoría de software. Entre los distintos formatos binarios, el *Executable and Linkable Format* (ELF) se ha consolidado como el estándar predominante en sistemas tipo UNIX, incluyendo la mayoría de las distribuciones de Linux. Sin embargo, entender y comparar funciones a partir de este tipo de binarios, sin contar con información simbólica (nombres originales de funciones, variables, estructuras de datos), continúa siendo un desafío técnico de gran complejidad [1].

El análisis de funciones sin símbolos es una tarea crítica cuando se trabaja con software compilado en entornos no controlados, como binarios extraídos de dispositivos embebidos, código dañino (*malware*), *firmware* o aplicaciones ofuscadas [22]. Este contexto impone la necesidad de desarrollar herramientas más inteligentes, capaces de identificar patrones estructurales y semánticos directamente desde el código binario [5].

En este Trabajo de Fin de Máster (TFM) se propone un enfoque que aprovecha modelos capaces de transformar cada función del ejecutable en una estructura de grafo denominada grafo de flujo de control (del inglés, *Control Flow Graph* o CFG), donde los nodos representan fragmentos indivisibles de instrucciones (bloques básicos), y las conexiones reflejan posibles transiciones en la ejecución del programa. Sobre estos grafos se entrenan redes neuronales que utilizan *autoencoders* multicapa y son capaces de generar representaciones internas compactas que capturan información relevante sobre la función [12, 13].

1.2. Objetivo del proyecto

El objetivo principal es desarrollar un sistema de aprendizaje automático capaz de clasificar funciones extraídas de binarios de Linux mediante *autoencoders* multicapa, representando dichas funciones como CFGs y generando *embeddings* comparables incluso entre arquitecturas distintas.

1.3. Estructura del documento

El documento se encuentra dividido en 6 capítulos y un anexo. En el Capítulo 2 se describen los fundamentos teóricos necesarios, incluyendo el análisis estático de binarios ELF, el uso de CFGs y la aplicación de redes neuronales sobre grafos. En el Capítulo 3 se presenta una revisión del estado del arte, destacando los enfoques existentes en comparación binaria. En el Capítulo 4 se detalla la propuesta del sistema con la metodología adoptada, incluyendo las herramientas utilizadas, el proceso de extracción de funciones y el diseño de los modelos. Se expone el desarrollo práctico del sistema, describiendo las fases de implementación y entrenamiento de los modelos. En el Capítulo 5 se presentan los resultados obtenidos en la evaluación del sistema, junto con un análisis crítico de su efectividad y limitaciones. En el Capítulo 6 se recogen las conclusiones del trabajo y se proponen líneas de mejora y exploración futura. Al final del documento se encuentra el Anexo A, donde se muestra cómo se ha distribuido el tiempo invertido en este trabajo.

Capítulo 2

Fundamentos teóricos

En este capítulo se presentan los fundamentos conceptuales necesarios para comprender el sistema propuesto, el cual combina técnicas de análisis estático de binarios ELF con modelos de aprendizaje automático sobre grafos aplicados a representaciones estructurales del código. Se abordan las bases del análisis de ejecutables, la representación de funciones como grafos, y los modelos de *autoencoders* (incluyendo variantes sobre grafos) que permiten extraer representaciones significativas para tareas de clasificación y agrupamiento.

2.1. Formato ELF y análisis estático

Executable and Linkable Format (ELF) es el formato estándar para archivos binarios ejecutables y bibliotecas compartidas en sistemas Unix y derivados, como Linux. Su diseño modular permite representar eficazmente instrucciones de código, datos, símbolos y estructuras auxiliares necesarias para la carga y ejecución del programa.

Un archivo ELF está compuesto por distintas partes: una cabecera principal, con metadatos del ejecutable; y una tabla de secciones, que segmenta el contenido en partes como `.text` (código), `.data` (datos inicializados), `.bss` (datos no inicializados), entre otras. Opcionalmente, se puede incluir una tabla de símbolos, que proporciona nombres y direcciones de funciones o variables (frecuentemente eliminada por razones de optimización o seguridad). En este TFM se va a tratar con binarios sin información simbólica (denominados *stripped* en inglés), lo que exige recuperar la información funcional directamente desde el código ensamblador.

El análisis de binarios puede abordarse desde dos perspectivas: el análisis estático, que se realiza sin ejecutar el código del programa; y el análisis dinámico, donde se requiere examinar el programa en tiempo de ejecución. Ambos enfoques son complementarios, pero en este trabajo se centra en el análisis estático.

Como se ha mencionado, el análisis estático es el proceso de examinar un programa sin ejecutarlo, es decir, mediante el estudio de su código binario. Este análisis permite identificar instrucciones y funciones, recuperar bloques básicos de código y estructuras de flujos de control, y construir representaciones estructurales como CFGs a partir de las relaciones entre los bloques de código binario [1]. Herramientas como `radare2` [17] son empleadas para automatizar el desensamblado, la detección de funciones y la exportación de CFGs.

2.2. Grafos de flujo de control

Un grafo de flujo de control (del inglés, *Control Flow Graph* o CFG), es una representación abstracta de la lógica de ejecución de un programa. Se define como un grafo dirigido tal que $G = (V, E)$, donde:

- Cada nodo $v \in V$ representa un bloque básico atómico, entendido como una secuencia

de instrucciones que se ejecutan de forma lineal, sin saltos intermedios ni interrupciones. Dentro de un bloque básico, el control de flujo entra por la primera instrucción y solo puede salir por la última, lo que garantiza su ejecución completa siempre que sea alcanzado.

- Cada arista $(v_i, v_j) \in E$ representa una posible transición de ejecución entre bloques (por ejemplo: un salto condicional, de llamada o de retorno).

Este tipo de representación lógica es especialmente robusta ante modificaciones sintácticas en el código ensamblador, ya que captura la estructura de control más que la forma superficial del código (sintaxis de las instrucciones individuales) [1].

2.3. Aprendizaje profundo y *autoencoders*

El aprendizaje profundo es una rama del aprendizaje automático que emplea redes neuronales de múltiples capas para modelar relaciones complejas entre los datos. A través de un proceso de entrenamiento supervisado o no supervisado, estas redes son capaces de extraer representaciones jerárquicas de los datos de entrada.

Una red neuronal profunda transforma un vector de entrada a través de múltiples capas de activación no lineales, produciendo como resultado una salida que puede usarse para tareas de clasificación, reconstrucción o generación de datos, entre otras.

Los *autoencoders* son modelos de redes neuronales utilizados para aprender representaciones compactas (llamados *embeddings*) de los datos. Están compuestos por dos partes:

- Un codificador (*encoder*), que transforma la entrada x en una representación del espacio latente en forma de vector z. El espacio latente se define como un subconjunto de menor dimensión que el espacio original de entrada, en el que se espera conservar las características más relevantes de los datos, descartando la información redundante o ruidosa.

- Un decodificador (*decoder*), que intenta reconstruir la entrada original x a partir de z, generando una salida $\hat{x}$.

En este TFM se van a emplear métricas para evaluar la similitud entre funciones como aplicación de los *autoencoders*. El entrenamiento del *autoencoder* consiste en minimizar una función de pérdida de reconstrucción, como el error cuadrático medio [6]:

$$\mathscr{L}_{recon}(x, \hat{x}) = \|x - \hat{x}\|^2.$$

Se aplica la distancia euclidiana o L^2 entre x y $\hat{x}$ en el espacio latente. Esta función fuerza al modelo a aprender una representación de z ($\hat{x}$) que retenga la información más significativa de x, ya que una codificación deficiente impedirá reconstruir la entrada con precisión.

Por otro lado, en la fase de clasificación supervisada sobre datos representados como grafos, se utiliza una función de pérdida distinta, adecuada para tareas multiclase: la entropía cruzada. Esta función mide la discrepancia entre la distribución de clases predicha por el modelo y la distribución verdadera (codificada como etiquetas), y se define como:

$$\mathscr{L}_{\text{CE}}(y, \hat{y}) = -\sum_{i=1}^{C} y_i \log \hat{y}_i,$$

donde C es el número de clases, y_i es la etiqueta real en formato *one-hot* (forma de representar una clase categórica como un vector, donde solo una posición del vector es 1 (la clase verdadera) y el resto son ceros), e $\hat{y}_i$ representa la probabilidad predicha para la clase i por el modelo. Esta función penaliza fuertemente las predicciones incorrectas cuando el modelo asigna alta confianza a una clase errónea, y es la más común en problemas de clasificación multiclase.

En este TFM, esta segunda función ha sido utilizada durante el entrenamiento del modelo, con el objetivo de maximizar la probabilidad de asignar correctamente cada dato a su clase correspondiente.

Esta estructura favorece la generalización del modelo y permite una exploración más efectiva del espacio latente, lo cual es valioso para representar funciones binarias que presentan pequeñas variaciones estructurales [2]. Son especialmente útiles para tareas como agrupamiento, generación y detección de anomalías.

Una vez entrenado el *autoencoder*, se pueden comparar las representaciones latentes z_1, z_2 de dos funciones para determinar su similitud. Una métrica especialmente útil en este contexto es la similaridad coseno, que mide la orientación entre dos vectores independientemente de su magnitud:

$$\text{sim}_{\cos}(z_1, z_2) = \frac{z_1 \cdot z_2}{\|z_1\| \, \|z_2\|}.$$

Este enfoque es adecuado cuando se busca comparar funciones semánticamente similares, aunque su representación numérica exacta varíe debido a diferencias de compilación o arquitectura, y resulta útil cuando se manejan *embeddings* normalizados.

2.4. Redes neuronales sobre grafos

Las redes neuronales sobre grafos (del inglés, *Graph Neural Networks* o GNNs), son modelos de aprendizaje profundo diseñados para operar directamente sobre datos estructurados como grafos, permitiendo aprender representaciones vectoriales de nodos, aristas o incluso grafos completos. A diferencia de las redes tradicionales, donde los datos tienen forma de vectores o matrices, en una GNN cada nodo aprende una representación basada en su estructura local y sus atributos.

Esta representación se actualiza de forma iterativa mediante una función de agregación que combina el estado actual de cada nodo del grafo con el de sus vecinos [23]. Las GNNs son especialmente adecuadas para datos con relaciones no lineales, como los CFGs.

Entre las variantes más utilizadas de GNNs se encuentran las redes convolucionales sobre grafos (del inglés, *Graph Convolutional Network* o GCN), donde la actualización de los nodos sigue un principio de convolución sobre grafos [10]. La fórmula general de una capa en una GCN es:

$$h_v^{(l+1)} = \sigma\left(\sum_{u \in \mathcal{N}(v)} \frac{W^{(l)} h_u^{(l)}}{\sqrt{d_v d_u}}\right),$$

donde:

- $h_v^{(l)}$ es el estado del nodo v en la capa l,
- $\mathcal{N}(v)$ representa el conjunto de los nodos vecinos (adyacentes) de v,
- $W^{(l)}$ son los pesos de la capa,
- σ es una función de activación (como ReLU).

Estas redes permiten modelar funciones enteras como grafos y aprender *embeddings* que capturan patrones estructurales en los CFGs, como bucles, condicionales o llamadas recursivas, sin necesidad de analizarlos de forma manual.

Los *Graph Autoencoders* (GAE) son extensiones de los *autoencoders* aplicados al dominio de los grafos. Se componen de:

- Un codificador basado en GCN que genera una representación latente Z.
- Un decodificador que intenta reconstruir las aristas del grafo original desde *embeddings* latentes, habitualmente mediante un producto escalar entre nodos:

$$\hat{A}_{ij} = \sigma(z_i^T z_j).$$

Esto mejora la generalización y la estructura de distribuciones del espacio latente, permitiendo una representación continua y significativa de los datos [11].

Al igual que con los *autoencoders* estándar, la similaridad coseno puede utilizarse como métrica en el espacio embebido para comparar funciones según sus vectores latentes z, permitiendo así detectar funciones similares aún si su implementación difiere.

Capítulo 3

Estado del arte

En este capítulo se revisan las principales líneas de investigación relacionadas con el análisis de funciones binarias, el aprendizaje de representaciones basadas en espacios latentes y la clasificación de código ensamblador. El objetivo es contextualizar el trabajo realizado dentro de los avances recientes basadas en aprendizaje automático sobre grafos como estudios sobre la disponibilidad y características de conjuntos de datos relevantes.

Una tarea clave en este ámbito es la identificación automática de funciones pertenecientes a bibliotecas, especialmente en contextos de análisis de seguridad e ingeniería inversa. Esta tarea se enmarca en el problema más amplio de la similitud entre funciones binarias, que ha recibido una atención creciente por parte de la comunidad científica. En particular, se ha evidenciado la necesidad de técnicas robustas frente a transformaciones sintácticas como la optimización del compilador, la arquitectura objetivo o la ofuscación del código. A continuación, se discute el estado del arte en este campo.

3.1. Modelos para comparación de funciones binarias

Antes del uso generalizado del aprendizaje automático, la comparación entre funciones se abordaba mediante técnicas heurísticas centradas en métricas del flujo de control, firmas estructurales o representaciones vectoriales derivadas del código ensamblador. Herramientas como `IDA Pro` o `BinDiff` aplican técnicas de comparación y emparejamiento sobre CFGs, mientras que otras propuestas como los métodos de *hashing* semántico generan descriptores mediante conteo de n-gramas de instrucciones, histogramas de llamadas o vectores sintácticos compactos [3,8,18].

Si bien estos métodos destacan por su eficiencia, son especialmente frágiles frente a modificaciones superficiales del código, como las introducidas por diferentes compiladores, arquitecturas, niveles de optimización o técnicas de ofuscación. Estas limitaciones han motivado el desarrollo de soluciones más robustas basadas en aprendizaje automático, capaces de capturar propiedades más invariantes de las funciones binarias.

Los enfoques recientes exploran el uso de redes neuronales para aprender representaciones latentes que conserven la semántica funcional de las funciones. Uno de los trabajos más influyentes es [13], donde revisan en profundidad cómo el aprendizaje automático ha transformado el campo del análisis de similitud binaria. En dicho trabajo se destacan tanto los modelos supervisados como los no supervisados, así como el papel central que desempeñan las representaciones gráficas intermedias, como los CFGs, para capturar la estructura semántica de las funciones.

Entre los modelos neuronales más prometedores se encuentran las redes neuronales de grafos, que permiten operar directamente sobre estructuras no euclidianas. Dentro de esta familia, en [12] se proponen las *Graph Matching Networks* (GMNs), modelos capaces de aprender funciones de similitud entre pares de grafos. Aunque no fueron diseñadas específicamente para funciones binarias, su capacidad de detectar correspondencias estructurales entre subgrafos las hace especialmente útiles para comparar CFGs.

3.2. Conjuntos de datos y herramientas para el entrenamiento

El diseño y la evaluación de sistemas de aprendizaje profundo para el análisis de funciones de aplicaciones requiere conjuntos de datos bien estructurados, diversos y suficientemente amplios. La disponibilidad de datos etiquetados y realistas es un factor determinante en el rendimiento y la generalización de los modelos. A continuación se presentan algunos de los recursos más representativos en este ámbito.

`BinKit` es una infraestructura modular diseñada para facilitar el análisis a gran escala de funciones individuales contenidas en ejecutables binarios [9]. Su objetivo principal es proporcionar un conjunto de herramientas que permitan desensamblar, analizar y comparar funciones de manera eficiente, incluso en escenarios donde los binarios están compilados estáticamente o con distintas *toolchains*.

Una de las principales fortalezas de `BinKit` es su capacidad para extraer funciones a nivel binario y asociarlas con metadatos relevantes, como su procedencia, tipo de biblioteca de origen o patrón de llamadas. Esta característica lo hace especialmente útil para tareas de clasificación o similitud de funciones, donde se desea distinguir entre código propio y funciones pertenecientes a bibliotecas estándar o de terceros.

El *pipeline* de `BinKit` incluye etapas de análisis como desensamblado, normalización de instrucciones, extracción de bloques básicos y generación de representaciones estructurales (por ejemplo, árboles o grafos). Esto permite su integración con arquitecturas modernas de aprendizaje automático, especialmente modelos orientados a grafos como GNNs o GMNs. Además, su diseño escalable permite generar conjuntos de datos de miles de funciones distribuidas en diferentes categorías, como compiladores (`GCC`, `Clang`), arquitecturas (x86, ARM) o tipos de bibliotecas utilizadas.

En investigaciones recientes, `BinKit` ha sido empleado como base para construir conjuntos de datos realistas que simulan entornos de análisis de malware, identificación de funciones de bibliotecas o detección de reutilización de código. Su modularidad también lo hace apto para incorporar nuevas etapas de análisis, como la anotación semántica de funciones o la extracción de rasgos sintácticos mediante embeddings. Es utilizado en trabajos como `Gemini` [21] y posteriores propuestas basadas en GNNs [12].

Self-Attentive Function Embeddings (SAFE) es un *framework* propuesto por Massarelli et al. [14] para la representación densa de funciones binarias mediante técnicas de aprendizaje profundo. Su principal contribución es el uso de un modelo entrenado sobre funciones extraídas de binarios, capaz de generar *embeddings* que capturan tanto rasgos sintácticos como semánticos, sin necesidad de información de alto nivel como el código fuente.

El *pipeline* de SAFE parte de la representación del código máquina de una función como una secuencia de instrucciones. A través de una combinación de capas de *embedding*, mecanismos de atención y redes neuronales recurrentes, el modelo produce vectores de longitud fija que codifican la función en un espacio latente, donde funciones similares quedan cercanas entre sí.

Esta representación puede ser utilizada para tareas que incluyen la medición de similitud binaria entre funciones compiladas con diferentes *toolchains* para distintas arquitecturas; la clasificación de funciones como propias o de bibliotecas; y el agrupamiento o búsqueda de funciones similares dentro de grandes colecciones de binarios.

Aunque el modelo original se entrenó con funciones extraídas mediante herramientas como `IDA Pro`, su arquitectura puede adaptarse a otras herramientas como `Angr` o `radare2`, siempre que se disponga de una secuencia representativa de instrucciones por función. En investigaciones recientes, SAFE ha sido combinado con arquitecturas gráficas como GMN o GCN para enriquecer aún más las representaciones estructurales y semánticas de funciones. Un ejemplo de ello es el desarrollo de modelos que incorporan explícitamente la estructura de los CFGs [7].

A pesar del notable progreso en el análisis automático de funciones de archivos binarios, sigue existiendo una carencia de estudios centrados en la clasificación supervisada de funciones

compiladas (particularmente en binarios sin información simbólica) en función de su procedencia (bibliotecas o paquetes). La mayoría de los trabajos revisados se orientan hacia la comparación entre pares de funciones o la medición de la similitud, dejando relativamente inexplorada la tarea de clasificación multiclase basada en representaciones estructurales. Asimismo, se observa una escasa integración práctica entre herramientas de análisis estático reales y arquitecturas de aprendizaje profundo sobre grafos.

Este trabajo se plantea como una contribución en esa dirección: se propone un sistema que parte del análisis estructural de binarios reales mediante herramientas como `radare2`, y se aplica modelos de grafos neuronales para clasificar funciones individuales según su origen. Al hacerlo, se busca cubrir el vacío existente entre la teoría de modelos de similitud y su aplicación práctica a tareas de clasificación funcional dentro de entornos reales y potencialmente adversos, como el análisis de *malware*.

Capítulo 4

Arquitectura del sistema propuesto

En este capítulo se describen en detalle los componentes, herramientas y etapas que conforman el sistema desarrollado para la clasificación automática de funciones de archivos binarios. Se detalla primero el flujo general del sistema y su organización en etapas, incluyendo los procesos de desensamblado, extracción de CFGs y su conversión a representaciones aptas para modelos de aprendizaje automático. Después, se presenta la estructura del modelo de aprendizaje utilizado, incluyendo la representación de atributos, la arquitectura de la red neuronal y las características del entrenamiento. Luego, se decribe el conjunto de herramientas utilizadas para el análisis y la transformación de binarios. Finalmente, se indican las técnicas adoptadas para facilitar su reproducibilidad y evaluación experimental.

4.1. Descripción del sistema

El sistema desarrollado tiene como objetivo clasificar funciones de archivos ejecutables de Linux en función de su procedencia (bibliotecas) mediante el uso de modelos de grafos. Para ello, se diseña un *pipeline* automatizado, reproducible y completamente implementado en Python, que transforma archivos binarios reales en representaciones de grafos aptas para modelos de aprendizaje profundo.

Este *pipeline* se ilustra en el Algoritmo 1. El proceso comienza a partir de una muestra del conjunto de datos `BinKit`, previamente organizada jerárquicamente por paquete, arquitectura, compilador y nivel de optimización. A partir de esta organización, se extrae un subconjunto curado de binarios en formato ELF, sobre el cual se ejecuta un análisis estático utilizando `radare2`, permitiendo obtener los CFGs de cada función presente en los binarios (líneas 1 a 7). Posteriormente, los CFGs se transforman en grafos dirigidos utilizando `NetworkX`, incorporando metadatos como el tamaño de los bloques, las relaciones de salto (directo, indirecto, condicional, llamadas externas o al sistema), así como nodos sintéticos para mantener conectividad (líneas 8 a 13). Este paso incluye mecanismos heurísticos para la recuperación de nodos aislados. `NetworkX` es una biblioteca de Python especializada en la creación, manipulación y análisis de grafos complejos, lo que permite estructurar los CFGs de forma flexible y eficiente, facilitando su posterior procesamiento en tareas de aprendizaje automático.

Una vez construidos, estos grafos se convierten a objetos de `Deep Graph Library`, una biblioteca específica para aprendizaje automático y análisis sobre grafos con atributos normalizados de Python, y se etiquetan automáticamente en función de su directorio de origen, que codifica el paquete de procedencia (líneas 14 a 17). El conjunto resultante se filtra para eliminar nodos desconectados, generando un conjunto de datos limpio y estructurado para tareas de aprendizaje supervisado.

El modelo entrenado es un clasificador GCN de dos capas, que opera sobre los atributos del grafo (en este caso, el tamaño de cada bloque). El objetivo es realizar clasificación multiclase de funciones, discriminando entre distintas bibliotecas o componentes software (en este caso, la

arquitectura).

El sistema entrena el modelo usando la función de pérdida de entropía cruzada, optimización con Adam y validación cruzada a través de múltiples ejecuciones con distintos valores de semillas (del 1 al 50), a fin de evaluar la robustez del enfoque. El conjunto de datos se divide de forma estratificada entre entrenamiento y test, y se analizan métricas estándar como exactitud, precisión, *recall*, *F1 score* y área bajo la curva ROC (AUC), además de tiempos de entrenamiento y evaluación (líneas 18 a 22).

Algorithm 1 Pipeline de clasificación de funciones de archivos binarios mediante GCN

Entrada: Directorio de binarios organizados por paquete y configuración
Salida: Métricas de rendimiento de clasificación multiclase

Etapa 1: Extracción de CFGs
1: **for all** binarios `.elf` en la estructura **do**
2: Ejecutar `aflq` para obtener direcciones de funciones
3: **for all** direcciones de funciones **do**
4: Ejecutar `agfj` para obtener CFG en formato JSON
5: **end for**
6: Guardar el conjunto de CFGs en `agfjAll/`
7: **end for**
Etapa 2: Parseo y representación como grafos
8: **for all** archivos JSON en `agfjAll/` **do**
9: Convertir a grafo dirigido en `NetworkX`
10: Añadir atributos semánticos (saltos, tamaños, llamadas externas)
11: Aplicar heurísticas para conectar nodos aislados
12: Guardar en `nxagfjAll/`
13: **end for**
Etapa 3: Conversión a DGL y etiquetado
14: **for all** grafos en `nxagfjAll/` **do**
15: Asignar etiqueta de clase según nombre de paquete
16: Convertir a objeto de `DGLGraph` con atributos normalizados
17: **end for**
Etapa 4: Entrenamiento y evaluación
18: **for all** semillas en {1, 2, 3... 50} **do**
19: Particionar el dataset de forma estratificada
20: Entrenar modelo GCN con entropía cruzada y optimizador Adam
21: Evaluar en conjunto de test: Exactitud, Precisión, *Recall*, *F1 score*, AUC
22: **end for**
Resultado: Métricas promedio y gráficas por semilla

Tabla 4.1: Resumen técnico del sistema propuesto

Aspecto	**Detalles**
Conjunto de datos base	Muestra de `BinKit` (organizada por paquete, arquitectura y *toolchain*)
Número de binarios con funciones analizados	576
Número de clases	5 (una por paquete o biblioteca), 10 (si se tiene en cuenta en cada biblioteca la arquitectura)
Formatos binarios	ELF
Extracción de funciones	`radare2` (`aflq`, `agfj`)
Conversión a grafos	`NetworkX` (con reconexión heurística de nodos)
Librería de aprendizaje de grafos	`DGL` (Deep Graph Library)
Modelo	GCN de 2 capas (GraphConv)
Tamaño del embedding oculto	16
Atributos usados por nodo	Tamaño del bloque básico
Atributos de las aristas	Tipo de salto (JUMP, FAIL, SYSCALL, etc.)
Pérdida	Entropía cruzada
Optimizador	Adam
Semillas usadas	{1, 2, 3... 50}
Epochs por entrenamiento	3000
Tamaño de batch	16
Partición del dataset	80 % entrenamiento / 20 % test (estratificada)
Métricas reportadas	Exactitud, Precisión, *Recall*, *F1 score*, AUC, tiempo de entrenamiento/evaluación

4.2. Implementación técnica

El sistema propuesto está construido sobre un conjunto de herramientas y bibliotecas especializadas ampliamente utilizadas en análisis binario, procesamiento de grafos y aprendizaje profundo. A continuación, se describen brevemente dichas herramientas.

`radare2` [17] es un *framework* de análisis binario estático de código abierto que permite inspeccionar y desensamblar binarios en múltiples arquitecturas. Se emplea para obtener los CFGs de cada función, mediante comandos como `aflq` (lista de funciones) y `agfj` (grafo de una función en formato JSON).

`NetworkX` [15] es una biblioteca de Python para la manipulación y análisis de grafos. Se utiliza para representar y procesar los CFGs extraídos, permitiendo enriquecerlos con atributos semánticos, normalizar su estructura y aplicar heurísticas para la reconexión de nodos aislados.

DGL [4] es un *framework* especializado en el aprendizaje automático sobre grafos, que permite transformar grafos de `NetworkX` a objetos optimizados para modelos como GCN o VGAE. Facilita el uso eficiente de redes neuronales sobre estructuras no euclidianas.

`PyTorch` [16] es una biblioteca de referencia para aprendizaje profundo en Python. DGL se apoya en `PyTorch` para definir y entrenar modelos como el clasificador GCN implementado en este proyecto.

Por último, `scikit-learn` [19] se emplea para la partición estratificada del conjunto de datos, así como para el cálculo de métricas estándar (exactitud, precisión, *recall*, *F1 score*, AUC) que permiten evaluar el rendimiento del modelo.

4.3. Reproducibilidad del sistema

La ejecución del sistema está organizada en etapas secuenciales: extracción de funciones, construcción de grafos, etiquetado, conversión a DGL, entrenamiento y evaluación, tal y como se ilustra en el Algoritmo 1. La arquitectura del código permite reproducir todos los experimentos desde cero. Además, se han generado automáticamente logs, métricas por semilla y gráficas de evolución del entrenamiento.

La Tabla 4.1 resume los aspectos técnicos más relevantes del sistema, incluyendo número de binarios procesados, número de clases, parámetros del modelo, atributos considerados y configuración del entrenamiento.

El sistema desarrollado genera automáticamente directorios de registros por cada ejecución del entrenamiento y la evaluación. Estos registros incluyen, para cada semilla utilizada, las métricas obtenidas (exactitud, precisión, *recall*, *F1 score*, AUC), así como los tiempos totales de entrenamiento y de evaluación. Además, se almacenan gráficos que representan la evolución de la función de pérdida y de cada una de estas métricas a lo largo de las iteraciones del modelo.

Para asegurar la reproducibilidad del sistema propuesto, se recomienda ejecutar el pipeline completo sobre un entorno Linux que cuente con `radare2` instalado y correctamente configurado. Es importante verificar que se disponen de los permisos adecuados para analizar los archivos binarios en formato ELF, especialmente si han sido generados o descargados externamente. Asimismo, se debe seguir una estructura de carpetas del conjunto de datos determinada, ya que esta estructura se utiliza para asignar automáticamente las etiquetas de clase durante el preprocesamiento. Por último, se aconseja realizar múltiples ejecuciones con distintas semillas aleatorias con el fin de obtener resultados estadísticamente representativos.

4.4. Limitaciones de la aproximación

Aunque el sistema propuesto ha demostrado ser efectivo en la clasificación de funciones de archivos binarios por su procedencia, presenta varias limitaciones que es importante reconocer.

En primer lugar, la calidad de los grafos obtenidos depende fuertemente de la precisión del análisis estático realizado por `radare2`. En binarios altamente optimizados u ofuscados, la extracción de funciones o bloques puede ser incompleta o incorrecta, lo cual afecta directamente la calidad del grafo construido.

En segundo lugar, la representación utilizada se basa únicamente en atributos estructurales, como el tamaño de los bloques o el tipo de salto, sin incluir aún información semántica más rica, como patrones de instrucciones o secuencias temporales. Esto limita la capacidad del modelo para distinguir funciones con estructura similar pero comportamiento funcional diferente.

Asimismo, el número de clases considerado se restringe a bibliotecas dentro del subconjunto de `BinKit` utilizadas. En un entorno real, donde los binarios pueden contener funciones procedentes de muchas fuentes heterogéneas y desconocidas, el modelo actual no se generalizaría sin un reentrenamiento exhaustivo.

Por último, el modelo GCN implementado es relativamente simple en cuanto a arquitectura. En trabajos futuros pueden explorarse variantes más avanzadas, como redes con mecanismos de atención o codificadores variacionales sobre grafos, que capturen mejor la variabilidad estructural y semántica del código binario.

Capítulo 5

Evaluación del sistema

Este capítulo describe la metodología experimental utilizada para evaluar el rendimiento del sistema propuesto, así como los resultados obtenidos en diferentes escenarios de clasificación de funciones binarias. El objetivo es analizar la precisión, estabilidad y robustez del modelo frente a variaciones tanto en la partición de los datos como en la configuración de clases. La evaluación se ha llevado a cabo de forma sistemática sobre un conjunto representativo de funciones extraídas de binarios reales.

5.1. Metodología experimental

Para medir el comportamiento del sistema en condiciones variadas, se ha optado por realizar múltiples ejecuciones independientes del modelo, utilizando distintas semillas aleatorias. Cada ejecución consiste en un entrenamiento completo del modelo GCN durante 3000 *epochs* (iteraciones completas por el conjunto de entrenamiento), seguido de una evaluación sobre un subconjunto de prueba. En total, se han realizado 50 ejecuciones, correspondientes a distintas semillas, con el objetivo de estimar la variabilidad del rendimiento frente a cambios aleatorios en la inicialización del modelo y la partición de los datos.

La partición del conjunto de datos se realiza de forma estratificada, reservando el 80 % para entrenamiento y el 20 % para evaluación. Al finalizar cada ejecución, se registran automáticamente las métricas de rendimiento, los tiempos de entrenamiento y evaluación, y se generan gráficas que permiten visualizar la evolución de cada métrica a lo largo de los *epochs*.

5.2. Métricas de evaluación utilizadas

Para evaluar el modelo se han utilizado métricas estándar en tareas de clasificación multiclase. Concretamente, exactitud, precisión, *recall*, *F1 score* y área bajo la curva, descritas a continuación.

La exactitud mide la proporción de predicciones correctas sobre el total de muestras evaluadas. Se define formalmente como:

$$\text{Exactitud} = \frac{\text{Número de predicciones correctas}}{\text{Número total de muestras}} = \frac{TP + TN}{TP + TN + FP + FN},$$

donde TP y TN representan los verdaderos positivos y verdaderos negativos, respectivamente, mientras que FP y FN corresponden a los falsos positivos y falsos negativos, respectivamente.

Aunque la exactitud es una métrica intuitiva y ampliamente utilizada, puede resultar poco representativa en contextos donde el conjunto de datos está desbalanceado, es decir, cuando algunas clases están sobrerrepresentadas con respecto a otras. En estos casos, un modelo puede obtener una alta exactitud simplemente prediciendo siempre la clase mayoritaria, sin captar la complejidad real del problema.

La precisión media (*macro precision*) se obtiene calculando la precisión por clase y promediando los resultados; la precisión de una clase i (Precision_i) se define como el cociente entre verdaderos positivos y la suma de verdaderos positivos y falsos positivos de la clase i:

$$\text{Precision}_i = \frac{TP_i}{TP_i + FP_i}$$

De manera análoga, el *macro recall* se obtiene promediando el *recall* de una clase i, que es la tasa de verdaderos positivos sobre el total de elementos de cada clase.

$$\text{Recall}_i = \frac{TP_i}{TP_i + FN_i}$$

El *macro F1 score* se calcula como la media armónica entre precisión y *recall*, también promediada entre clases. De esta forma se obtiene una métrica balanceada entre la precisión y el *recall*, al ponderar estas dos. Formalmente:

$$\text{F1}_i = 2 \cdot \frac{\text{Precision}_i \cdot \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}$$

Por último, se incluye el área bajo la curva ROC (AUC-ROC), empleando la estrategia uno contra el resto para cada clase, que permite evaluar la capacidad discriminativa del modelo incluso en escenarios con clases desequilibradas.

Para cada ejecución también se calcula una matriz de confusión, la cual permite visualizar con detalle las confusiones (errores del modelo) entre clases específicas.

5.3. Resultados experimentales

El sistema ha sido evaluado en dos configuraciones distintas: una primera con cinco clases, correspondientes a diferentes bibliotecas de software; y una segunda más exigente con diez clases, donde cada clase representa una combinación específica de biblioteca y arquitectura de compilación. En la Tabla 5.1 se muestra un resumen comparativo de las métricas obtenidas en ambas configuraciones.

En el escenario con cinco clases, el modelo alcanzó una exactitud media del 75.88 %, una precisión media del 80.51 %, un *recall* medio del 80.13 %, y un *F1 score* medio del 79.24 %. El área bajo la curva ROC fue de 0.9427. El tiempo medio de entrenamiento por ejecución fue de 740.32 segundos, mientras que el tiempo medio de evaluación fue de tan solo 0.048 segundos. Sumando las cincuenta ejecuciones, el tiempo total invertido en entrenamiento asciende a aproximadamente 10.28 horas, con una evaluación acumulada de 2.39 segundos.

En el segundo escenario, con diez clases, los resultados reflejan una mayor complejidad en la tarea de clasificación. La exactitud media se situó en el 65.29 %, la precisión media fue del 72.15 %, el *recall* del 68.94 % y el *F1 score* alcanzó el 68.54 %. El valor medio de AUC-ROC en esta configuración fue ligeramente superior al caso anterior, con un valor de 0.9473. Los tiempos promedio de entrenamiento y evaluación fueron de 818.58 y 0.054 segundos, respectivamente, lo que representa un total aproximado de 11.37 horas de entrenamiento y 2.68 segundos de evaluación.

Tabla 5.1: Resumen de resultados experimentales del modelo GCN para 5 y 10 clases

Métrica	**5 clases**	**10 clases**
Exactitud media (%)	75.88	65.29
Precisión media (%)	80.51	72.15
Recall media (%)	80.13	68.94
F1-score media (%)	79.24	68.54
AUC-ROC promedio	0.9427	0.9473
Tiempo medio entrenamiento (s)	740.32	818.58
Tiempo medio evaluación (s)	0.048	0.054

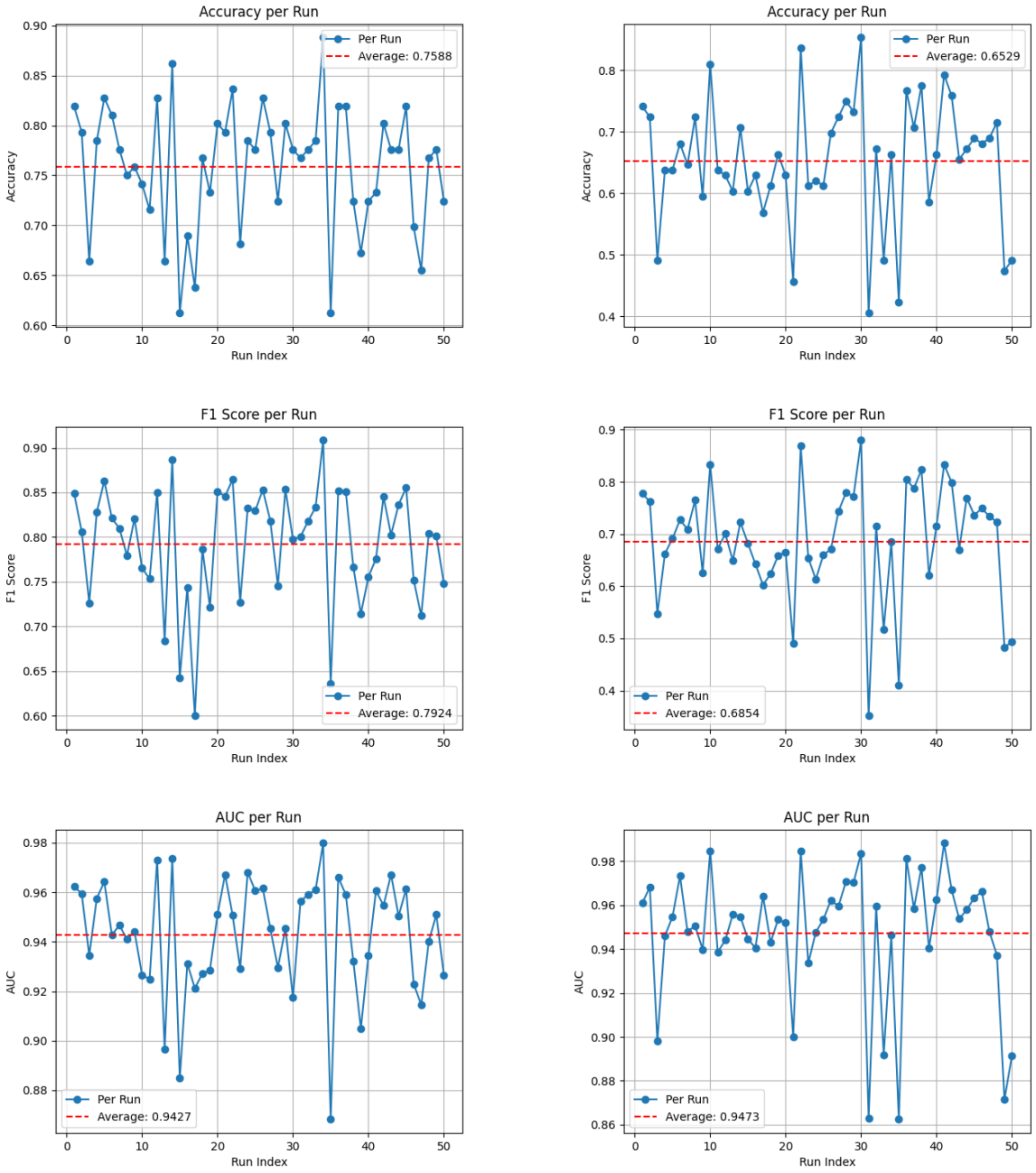

Figura 5.1: Evolución de las métricas durante el entrenamiento para las configuraciones de 5 clases (izquierda) y 10 clases (derecha). Se muestran: exactitud (arriba), *F1 score* (centro) y AUC (abajo).

5.4. Discusión de resultados

Además de las métricas cuantitativas, en la Figura 5.1 se presentan las gráficas de evolución de la exactitud, el *F1 score* y el AUC durante el entrenamiento en ambas configuraciones, utilizando 50 semillas distintas. Estas curvas permiten visualizar la estabilidad del modelo y la consistencia de su rendimiento a lo largo del entrenamiento.

Las gráficas muestran una tendencia estable y consistente en la mayoría de las ejecuciones, lo cual refuerza la validez del modelo y su capacidad para generalizar. Las variaciones entre semillas son moderadas, lo que indica que la arquitectura del sistema es robusta frente a la aleatoriedad introducida por las particiones de los datos.

Los resultados obtenidos validan la eficacia del modelo basado en GCN para clasificar funciones de archivos binarios según su origen. En la configuración de cinco clases se obtiene un rendimiento notable, con un *F1 score* superior al 79 % y una capacidad discriminativa muy alta, medida por AUC. En la configuración de diez clases, el sistema mantiene un comportamiento competitivo, con métricas por encima del 68 % incluso en un escenario con mayor complejidad y más clases cercanas entre sí. Hay que tener en cuenta que estos resultados han sido obtenidos combinando arquitecturas en el mismo espacio latente, no con *autoencoders* independientes por arquitectura. Por eso, los resultados de esta primera aproximación pueden considerarse competitivos.

La ligera caída en la precisión es razonable, dado que muchas funciones compiladas para diferentes arquitecturas presentan patrones estructurales similares. Aún así, el modelo es capaz de captar diferencias suficientes para mantener una buena separación entre clases. Los tiempos de ejecución también son razonables, lo cual refuerza la viabilidad del enfoque en contextos prácticos. Otro factor limitante ha sido el conjunto de datos seleccionado, donde existen algunas familias muy cercanas, algo que no suele verse en otros trabajos similares, pero que nos ofrece una validación más realista de los resultados.

En conjunto, el sistema demuestra ser una solución eficaz y reproducible para la clasificación estructural de funciones binarias, especialmente en entornos donde no se dispone de información simbólica y es necesario distinguir automáticamente entre funciones propias y de biblioteca.

Capítulo 6

Conclusiones

Este trabajo ha desarrollado un sistema completo y reproducible para la clasificación de funciones de archivos binarios en función de su procedencia, combinando técnicas de análisis estático, modelado gráfico y aprendizaje profundo. A partir de una muestra curada del conjunto de datos `BinKit`, se diseñó un proceso automatizado que permite extraer funciones desde binarios en formato ELF, representarlas como grafos de flujo de control enriquecidos semánticamente, y utilizarlas como entrada para modelos convolucionales sobre grafos. El entrenamiento fue evaluado bajo 50 semillas distintas para estudiar la estabilidad del modelo y su capacidad de generalización. Los resultados experimentales alcanzaron métricas satisfactorias, con un *F1 score* medio superior al 79 % en escenarios de clasificación por biblioteca, y un rendimiento competitivo en configuraciones más exigentes con 10 clases (biblioteca y arquitectura combinadas), lo que valida la viabilidad del enfoque propuesto para entornos reales donde no se dispone de información simbólica.

Como trabajo futuro, se identifican varias direcciones de mejora y ampliación del sistema. En primer lugar, se plantea abordar escenarios de clasificación más complejos, como la identificación de funciones compiladas con técnicas de ofuscación o provenientes de bibliotecas desconocidas, lo cual sería útil en el contexto de análisis de *malware*. También se propone explorar modelos auto-supervisados y variantes probabilísticas como VGAE, que podrían generar representaciones latentes más informativas y robustas frente a transformaciones sintácticas. Asimismo, resulta de interés comparar arquitecturas más expresivas como *Graph Attention Networks* (GAT) o *Graph Isomorphism Networks* (GIN), que han demostrado mejorar la capacidad de modelado estructural en otras tareas de clasificación sobre grafos.

Otra línea futura consiste en ampliar la base de datos de entrenamiento con funciones extraídas de binarios más variados, incluyendo compilaciones reales de proyectos, así como muestras representativas de código malicioso o empaquetado. Esto permitiría validar la capacidad del sistema para generalizar fuera del entorno controlado de `BinKit` y enfrentarse a desafíos reales en seguridad informática. Finalmente, integrar el sistema en herramientas de análisis binario ya existentes, como `radare2`, podría facilitar su aplicación práctica por analistas y profesionales del ámbito forense.

Bibliografía

[1] Dennis Andriesse. *Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly.* No Starch Press, 2018.

[2] Yankun Chen, Jingxuan Liu, Lingyun Peng, Yiqi Wu, Yige Xu, and Zhanhao Zhang. Auto-Encoding Variational Bayes. *Cambridge Explorations in Arts and Sciences*, 2(1), 2024.

[3] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of Binaries through re-Optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–94, 2017.

[4] DGL Team. Deep Graph Library. `https://www.dgl.ai/`, 2024.

[5] Irfan Ul Haq and Juan Caballero. A Survey of Binary Code Similarity. *ACM Computing Surveys (CSUR)*, 54(3):1–38, 2021.

[6] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504–507, 2006.

[7] Suguru Horimoto, Keane Lucas, and Lujo Bauer. Approach for the Optimization of Machine Learning Models for Calculating Binary Function Similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329. Springer, 2024.

[8] He Huang, Amr M Youssef, and Mourad Debbabi. BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 155–166, 2017.

[9] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. *IEEE Transactions on Software Engineering*, 49(4):1661–1682, 2022.

[10] Thomas N Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907*, 2016.

[11] Thomas N Kipf and Max Welling. Variational Graph Auto-Encoders. *arXiv preprint arXiv:1611.07308*, 2016.

[12] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph Matching Networks for Learning the Similarity of Graph Structured Objects. In *International Conference on Machine Learning*, pages 3835–3845. PMLR, 2019.

[13] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How Machine Learning Is Solving the Binary Function Similarity Problem. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2099–2116, 2022.

[14] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, pages 309–329. Springer, 2019.

[15] NetworkX Developers. NetworkX – NetworkX documentation. `https://networkx.org/`, 2024.

[16] PyTorch. PyTorch. `https://pytorch.org/`, 2025.

[17] radareorg. radare2: Unix-like reverse engineering framework and command-line toolset. `https://github.com/radareorg/radare2`, 2024. GitHub repository.

[18] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 76–85, 2003.

[19] scikit-learn Team. scikit-learn: machine learning in Python – scikit-learn 1.7.0 documentation. `https://scikit-learn.org/`, 2025.

[20] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. SOK:(State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[21] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

[22] Ilsun You and Kangbin Yim. Malware Obfuscation Techniques: A Brief Survey. *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pages 297–300, 2010.

[23] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020.

Apéndice A

Planificación temporal y dedicación horaria

En este anexo se presenta un resumen del tiempo dedicado a cada una de las fases del proyecto desarrollado durante las prácticas, tanto desde el punto de vista cronológico como de la carga de trabajo invertida.

Cabe destacar que el desarrollo de este trabajo ha estado vinculado parcialmente a unas prácticas curriculares, realizadas dentro del grupo de investigación DisCo, bajo la supervisión del tutor y director de este trabajo, y financiadas por una beca de Trabajo de Fin de Máster del Instituto de Investigación en Ingeniería de Aragón. No obstante, se hace explícita la distinción entre las tareas realizadas en el marco de las prácticas curriculares (primeras 225 horas, correspondientes al periodo comprendido entre noviembre y finales de enero) y aquellas desarrolladas posteriormente en el contexto exclusivo del TFM.

El cronograma mostrado en la Figura A.1 refleja la planificación real seguida durante las prácticas, incluyendo superposición de tareas y ajustes temporales realizados en función de los resultados obtenidos y la disponibilidad técnica.

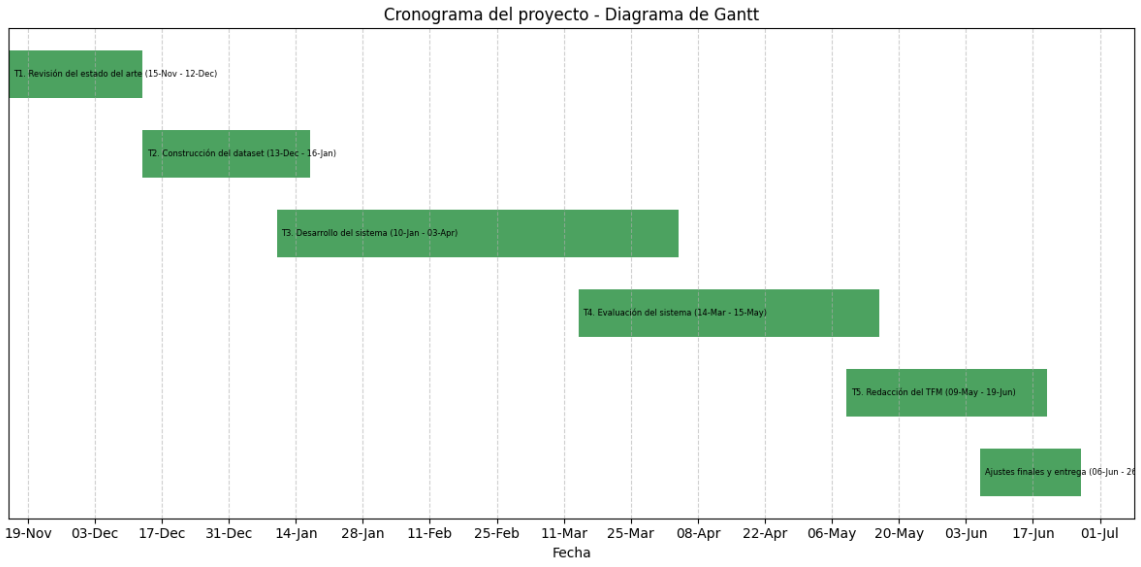

Figura A.1: Cronograma del proyecto

En la Tabla A se recoge la estimación de horas dedicadas a cada una de las tareas planificadas en el proyecto, de acuerdo con el reparto efectivo del tiempo a lo largo de las semanas. Las estimaciones reflejan tanto el desarrollo técnico como las reuniones de seguimiento, los ajustes realizados, y el tiempo invertido en redacción, revisión de resultados y mejora del sistema.

Tabla A.1: Horas estimadas por tarea

Tarea	**Descripción**	**Horas estimadas**
T1	Revisión del estado del arte y análisis bibliográfico inicial	75
T2	Construcción del conjunto de datos: recopilación, compilación, organización	80
T3	Desarrollo del sistema de análisis y clasificación de funciones	200
T4	Evaluación del sistema y análisis de resultados	150
	Reuniones	25
T5	Redacción del TFM y documentación de resultados	80
T6	Ajustes finales, generación de gráficas y entrega final	15
Total		**625 horas**