universidad de león

Departamento de Matemáticas

MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN CIBERSEGURIDAD

Trabajo de Fin de Máster

Comparación de funciones a través del análisis del grafo de control de flujo

Function Comparison Through Control Flow Graph Analysis

Autor: Carmen Carrero Hurtado

Tutor: Camino Fernández Llamas

Cotutor: Ricardo Julio Rodríguez Fernández

(Septiembre, 2023)

UNIVERSIDAD DE LEÓN
Departamento de Matemáticas
MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN CIBERSEGURIDAD
Trabajo de Fin de Máster

ALUMNO: Carmen Carrero Hurtado
TUTOR: Camino Fernández Llamas
COTUTOR: Ricardo Julio Rodríguez Fernández

TÍTULO: Comparación de funciones a través del análisis del grafo de control de flujo

TITLE: Function Comparison Through Control Flow Graph Analysis

CONVOCATORIA: Septiembre, 2023

RESUMEN:

En la actualidad, el análisis de código dañino (*malware*) y la detección de código compartido en diferentes aplicaciones supone un reto para los investigadores y profesionales del ámbito de la ciberseguridad. En este contexto, las técnicas de búsqueda aproximada, que permiten buscar una coincidencia en un valor próximo, combinadas con los algoritmos de similitud aproximada, se vuelven especialmente relevantes ya que permiten realizar análisis rápidos y escalables en grandes conjuntos de datos, facilitando así la detección y clasificación eficiente de nuevas variantes y familias de amenazas. En este Trabajo de Fin de Máster se ha desarrollado una herramienta que permite comparar y analizar ficheros ejecutables en busca de similitudes en las funciones que incorporan, utilizando el algoritmo de similitud aproximada *TLSH*. Esta herramienta permite identificar funciones similares entre diferentes binarios, resultando útil para encontrar código compartido o duplicado, detectar *malware* o ayudar en el análisis forense. Con el objetivo de verificar este sistema se han desarrollado y ejecutado una serie de pruebas para garantizar su robustez, versatilidad y buen funcionamiento. Los resultados de las pruebas realizadas muestran que la utilización de *TLSH* para calcular hashes de funciones basadas en bloques básicos del CFG es una técnica eficiente para detectar similitudes entre binarios.

Palabras clave: grafo de control de flujo, algoritmos de similitud aproximada, TLSH, CFG, análisis de binarios

Firma del alumno:	**VºBº Tutor:**

Abstract

Nowadays, the analysis of malicious code (*malware*) and the detection of shared code in different applications is a challenge for researchers and professionals in the field of cybersecurity. In this context, approximate search techniques, which allow to search for a match in a close value, combined with approximate similarity algorithms, become particularly relevant as they enable fast and scalable analyses on large datasets, thus facilitating the efficient detection and classification of new variants and threat families. In this Master's Thesis, we develop a tool to compare and analyse executable files in search of similarities in the functions they incorporate, using the *TLSH* approximate similarity algorithm. This tool allows to identify similar functions between different binaries, being useful to find shared or duplicated code, detect malware, or help in forensic analysis. In order to verify this system, a series of tests are developed and executed to ensure its robustness, versatility and good performance. The results of the tests carried out show that the use of *TLSH* to calculate hashes of functions based on basic blocks of the CFG is an efficient technique for detecting similarities between binaries.

Índice general

Abstract **3**

Índice de figuras **IV**

Índice de tablas **V**

1. Introducción **1**

1.1. Motivación 1
1.2. Objetivos 2
1.3. Estructura del documento 2

2. Conceptos previos **3**

2.1. Análisis mediante control de flujo 3
2.2. Algoritmos de similitud aproximada 5

3. Análisis, diseño e implementación **8**

3.1. Requisitos del sistema 8
3.1.1. Requisitos funcionales 8
3.1.2. Requisitos no funcionales 9
3.2. Arquitectura del sistema 9
3.2.1. Diagrama de clases UML 9
3.2.2. Diagrama de secuencia UML 10
3.2.3. Diagrama de flujo UML 11
3.3. Funcionamiento y características 12
3.3.1. Investigación realizada 12
3.3.2. Tecnologías y herramientas utilizadas 15
3.4. Limitaciones 16
3.5. Disponiblidad del código 17

4. Evaluación y discusión de resultados **18**

4.1. Evaluación 18
4.1.1. Definición del entorno 19
4.1.2. Conjunto de datos para pruebas 19
4.2. Discusión de resultados 20
4.2.1. Número de funciones encontradas en los binarios 20
4.2.2. Número de funciones similares encontradas en los binarios mediante búsqueda basada en vecinos 21
4.2.3. Número de funciones similares encontradas en los binarios mediante búsqueda basada en umbral 23

5. Conclusiones, problemas encontrados y trabajo futuro 24
5.1. Conclusiones 24
5.2. Problemas encontrados 25
5.3. Trabajo futuro 25

Bibliografía 26

A. Seguimiento de proyecto fin de máster 29

Índice de figuras

2.1. Ejemplo de CFG de la función `main` 4

3.1. Diagrama de clases UML. 10
3.2. Diagrama de secuencia UML del caso de uso: obtener los hashes de un binario. 11
3.3. Diagrama de flujo UML con el proceso añadir hashes a la base de datos. 12

4.1. Número de funciones totales obtenidas por comandos (F_{com}) comparadas con el número de funciones totales obtenidas con la herramienta desarrollada (F_{herr}) 20
4.2. Número de funciones "verdad absoluta" (F_{va}) comparadas con el número de funciones similares obtenidas con búsqueda basada en vecinos (F_{k2}, F_{k5}, F_{k10}) 21
4.3. Número de funciones "verdad absoluta" (F_{va}) comparadas con el número de funciones similares obtenidas con búsqueda basada en vecinos (F_{k2}, F_{k5}, F_{k10}) (con *ef* =4) 22
4.4. Número de funciones "verdad absoluta" (F_{va}) comparadas con el número de funciones similares obtenidas con búsqueda basada en umbral (F_{p60}, F_{p75}, F_{p90}) 23

A.1. Diagrama de Gantt. 30

Índice de tablas

3.1. Herramientas analizadas 13
3.2. Algoritmos y heurísticas utilizados por las herramientas estudiadas para la creación de CFGs 14

A.1. Número de horas trabajadas 30

Capítulo 1

Introducción

1.1. Motivación

En el mundo digital actual, donde las amenazas a la seguridad informática evolucionan constantemente, la capacidad de comprender el comportamiento del software que se ejecuta es esencial para garantizar la integridad y seguridad de los sistemas. Trabajar con código binario (resultado final de una compilación o fichero objeto) es esencial, puesto que el código fuente del software no siempre está disponible para su análisis.

Los enfoques de similitud de código binario comparan dos o más fragmentos de código, como bloques básicos, funciones o programas completos, para identificar sus similitudes y diferencias [17]. La comparación de código binario se convierte en fundamental en escenarios donde el código fuente del programa no está disponible, lo que sucede en programas comerciales o en código dañino (*malware*). La similitud de código binario tiene una amplia lista de aplicaciones en el mundo real, como la búsqueda de bugs [13, 14] detección de *malware* [11, 12] o detección de robo de propiedad intelectual en software [18], entre otras.

Una de las herramientas más potentes para el análisis de aplicaciones binarias es el análisis mediante grafos de control de flujo (CFG) [9]. Este tipo de análisis proporciona una visión profunda y estructural de cómo interactúan y se comportan las funciones dentro de un programa, lo que permite identificar patrones y similitudes de manera eficiente y precisa.

Actualmente, existen diversas herramientas de desensamblado que permiten mostrar las instrucciones que ejecuta el procesador. Estas instrucciones se muestran en

lenguaje ensamblador. Sin embargo, este lenguaje es difícil de leer y entender. Este trabajo surge de la necesidad de comprobar si determinadas funciones (como por ejemplo, funciones asociadas a *malware*) existen dentro de un binario sin que el profesional tenga que utilizar herramientas de desensamblado en el proceso.

1.2. Objetivos

El propósito de este Trabajo de Fin de Máster(TFM) es comparar funciones de aplicaciones binarias mediante análisis de CFGs. Para lograr este objetivo se ha construido un sistema, basado en la herramienta de análisis de aplicaciones *angr* [27], que analiza el CFG de un binario dado y obtiene los hashes de sus funciones, almacenándolas en una base de datos. El sistema también permite comparar la información contenida en esta base de datos con las funciones de otro binario, devolviendo aquellas que se consideran similares según un cierto umbral. Para la experimentación, se han considerado las aplicaciones de `binutils` [5] y una base de datos conformada por hashes de la aplicación `strings`, que forma parte de la misma colección de herramientas.

1.3. Estructura del documento

Este documento se estructura en 6 capítulos y un anexo. El *Capítulo* 2 expone los conceptos previos, centrándose en el análisis mediante control de flujo y los algoritmos de similitud aproximada. El *Capítulo* 3 describe el análisis, diseño e implementación del sistema desarrollado. En el *Capítulo* 4 se muestran y discuten los resultados obtenidos tras la experimentación. El *Capítulo* 5 presenta las conclusiones y propone una serie de ampliaciones e investigaciones futuras.

La memoria también cuenta con un *Anexo* A en el que se muestra un desglose y distribución del tiempo empleado en la realización de este TFM, así como las fases del mismo explicadas brevemente.

Capítulo 2

Conceptos previos

Este capítulo desarrolla en primer lugar el análisis mediante control de flujo. Después, explica los algoritmos de similitud aproximada y su funcionamiento.

2.1. Análisis mediante control de flujo

Un grafo de control de flujo (CFG; del inglés, *control flow graph*) es una representación esquemática en forma de grafo dirigido, de todos los caminos que pueden ser atravesados por un programa durante su ejecución. En un CFG, los nodos representan bloques básicos de la aplicación, mientras que los arcos representan los distintos caminos de ejecución que puede seguir el programa [9]. Un bloque básico es una secuencia lineal de instrucciones de un programa que tiene un único punto de entrada (la primera instrucción ejecutada) y un único punto de salida (la última instrucción ejecutada).

Por ejemplo, teniendo en cuenta el programa mostrado en el Código 2.1, se obtiene el CFG de la Figura 2.1 para la función `main`. El grafo obtenido cuenta con 9 bloques básicos. El primer bloque (punto de entrada 0x401149) acaba con una instrucción de ensamblador de Intel x86 que representa un salto condicional, la instrucción `jne`. Esto representa la evaluación de la variable number que aparece en la línea 4 del Código 2.1. Si se cumple la condición, el flujo continúa por el bloque básico cuyo punto de entrada es 0x401178 (que representa la línea 6 del Código 2.1). En caso contrario, continúa por el bloque básico cuyo punto de entrada es 0x401162 (línea 8 del Código 2.1). Ambos bloques acaban con una llamada a la función `printf` (bloque 0x401050). Tras estas ejecuciones, el flujo de ejecución finaliza en el bloque con punto de entrada 0x40118c, que representa el final de la función `main` (línea 10

del Código 2.1).

```c
#include <stdio.h>

int main() {
  int number = 5;
   if(number == 6) {
     printf("Number is: 6");
   } else {
     printf("Number is: 5");
   }
  return 0;
}
```

Código 2.1: Código C utilizado para obtener el CFG de ejemplo de la Figura 2.1

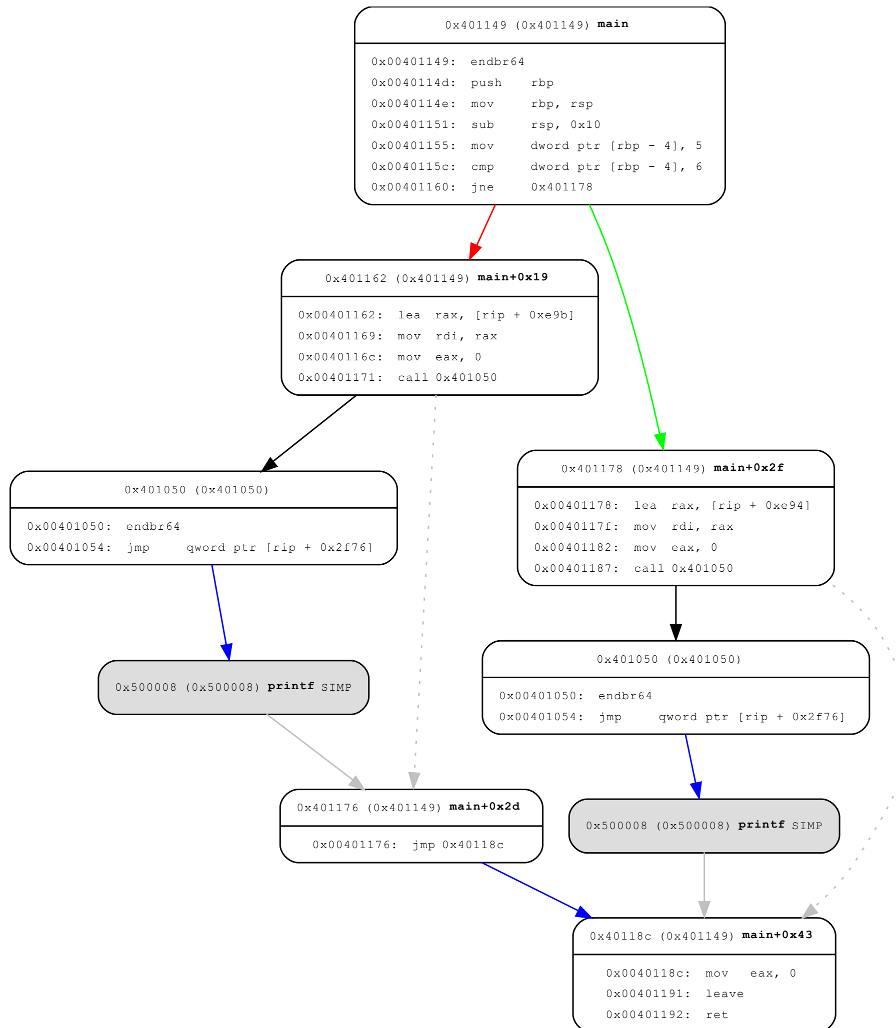

Figura 2.1: Ejemplo de CFG de la función `main`

Un análisis mediante CFG implica el estudio exhaustivo de cómo fluye la ejecución de un programa a través de su código binario. Esta técnica es esencial para comprender cómo se toman decisiones, qué caminos de ejecución pueden seguirse y cómo interactúan diferentes partes del programa. En resumen, un CFG representa visualmente las instrucciones del programa y las relaciones entre ellas, mostrando cómo se ramifican y cómo se conectan las diferentes partes del código.

2.2. Algoritmos de similitud aproximada

Los algoritmos de similitud aproximada son una técnica de *hashing* diseñada para identificar similitudes entre dos artefactos digitales [10]. Un artefacto digital se define como una secuencia arbitraria de bytes que tiene alguna interpretación significativa [10].

Esta identificación de similitudes entre dos o más artefactos puede realizarse en tres niveles diferentes de abstracción: *bytewise*, cuando la comparación se basa en la secuencia de bytes que forman el archivo; *sintánticos*, cuando se utilizan las estructuras internas de los archivos en lugar de meras secuencias de bytes; o *semánticos*, cuando la comparación se basa en atributos contextuales [20]. Además, estos algoritmos pueden comparar artefactos directamente o convertirlos primero en una representación intermedia que luego comparan. Esta representación intermedia es lo que se conoce como *digest*, cuyos algoritmos se denominan *Similarity Digest Algorithms* (SDA) o algoritmos de similitud de hashing o de *digest*. El objetivo de estos algoritmos es identificar objetos similares en lugar de objetos completamente idénticos, como hacen las funciones hash criptográficas [25]. Por lo tanto, los algoritmos de similitud de *digest* son un subconjunto de los algoritmos de similitud aproximada.

Basado en los estudios de distintos autores, los algoritmos existentes se pueden clasificar en las siguientes categorías [15, 16, 22]:

- **Statistically-improbable features:** abarca algoritmos que emplean una función de selección fundamentada en rasgos que son poco probables desde una perspectiva estadística.
- **Block-based hashing:** incluye algoritmos que hacen uso de funciones hash criptográficas para generar y guardar características para cada bloque de tamaño constante.

- **Block-based rebuilding:** consiste en algoritmos que eligen bloques (aleatoriamente seleccionados o prefijados) y generan los *digests* seleccionando los bloques más similares con respecto a la entrada.
- **Locality-sensitive hashing:** incluye algoritmos que asignan objetos en compartimentos, de manera que objetos similares tienden a agruparse en el mismo compartimento con alta probabilidad.

Sin embargo, estas categorías solo consideran aspectos específicos de los algoritmos, en vez de una visión completa del comportamiento. Esto puede producir malentendidos en cómo funcionan los algoritmos, lo que a la larga puede llevar a decisiones erróneas a la hora de seleccionar un SDA para un fin específico. Por ello, se ha propuesto otra clasificación nueva y más sencilla basada en el comportamiento completo [20]:

- **Feature Sequence Hashing:** contiene los algoritmos que dividen la entrada en características y las asignan midiendo la similitud mediante secuencias de características.
- **Byte Sequence Existence:** abarca los algoritmos que identifican la existencia (o similitud) de secuencias de bytes (llamadas bloques) en la entrada. El puntaje de similitud se calcula comparando el número de bloques comunes entre los resúmenes de similitud.
- **Locality-Sensitive Hashing:** engloba algoritmos que asignan elementos en contenedores, logrando agrupar elementos similares en el mismo contenedor con una alta probabilidad.

Un ejemplo de los algoritmos de similitud de *digest* es *TLSH* (Trend Micro Locality Sensitive Hash) [23]. Este algoritmo está basado en la técnica LSH para generación de hashes y está diseñado específicamente para la detección de similitud en archivos y datos, con un enfoque particular en la detección de código dañino (*malware*) y análisis forense de archivos.

Este algoritmo destaca por su escalabilidad, así como su enfoque en calcular hashes que reflejen la similitud en la estructura y distribución de los datos en lugar de su contenido específico. Esto lo hace útil para identificar archivos o bloques de datos que tienen patrones similares, incluso si los contenidos son diferentes. También hay que recalcar su rapidez, ya que el cálculo del hash *TLSH* es relativamente rápido

y eficiente en comparación con otros algoritmos. Además, es significativamente más difícil de atacar o evadir que otros algoritmos [21].

Capítulo 3

Análisis, diseño e implementación

Este capítulo se estructura en tres etapas que abordan los componentes técnicos esenciales para llevar a cabo el sistema propuesto en este trabajo. Durante la fase de análisis, se establecen los requisitos del sistema. Luego, en la etapa de diseño se configura la arquitectura a desarrollar. Por último, la etapa de implementación detalla el procedimiento seguido para implementar el sistema en su conjunto, describiendo las tecnologías empleadas, las limitaciones encontradas y el repositorio de software creado.

3.1. Requisitos del sistema

En este apartado se identifican los requisitos del sistema, separándolos en funciones y no funcionales. Estos requisitos son usados para la definición de la arquitectura usada en el diseño del proyecto.

3.1.1. Requisitos funcionales

- **RF1:** El sistema muestra un mensaje de ayuda al usuario con las opciones disponibles y el formato correcto para lanzar la ejecución del programa.
- **RF2:** El sistema permite la posibilidad de añadir nuevos hashes *TLSH* a la base de datos.
- **RF3:** El sistema guarda la base de datos en un almacenamiento secundario para su posterior uso.
- **RF4:** El sistema posee la capacidad de restaurar la base de datos desde un almacenamiento secundario.

- **RF5:** El sistema permite la posibilidad de realizar búsquedas de similitud aproximada en un binario cuyo nombre es pasado por parámetro y en función de un umbral de similitud definido por el usuario.

3.1.2. Requisitos no funcionales

- **RNF1:** El sistema consiste en una aplicación de línea de comandos.
- **RNF2:** El sistema puede manejar grandes volúmenes de datos y realizar búsquedas eficientes en tiempos razonables.
- **RNF3:** El sistema trabaja con ficheros binarios compilados para arquitectura de 64-bit de Intel.
- **RNF4:** El sistema usa *angr* [1] como herramienta de análisis de binarios.

3.2. Arquitectura del sistema

Con el objetivo de comprender la arquitectura del sistema se han realizado diagramas UML (*Unified Modeling Language*) [26]. UML es un lenguaje de modelo estandarizado que permite especificar, visualizar, construir y documentar los artefactos de los sistemas de software. Concretamente, se ha realizado el diseño del sistema mediante un diagrama de clases, un diagrama de secuencia y un diagrama de flujo, presentados a continuación.

3.2.1. Diagrama de clases UML

Un diagrama de clases UML representa un modelo conceptual de las clases y las relaciones existentes entre las mismas; es decir, permite tener una visión global de todo el sistema junto con todos sus componentes. El diagrama de clases del sistema desarrollado se visualiza en la Figura 3.1.

La arquitectura está conformada por varias clases. La clase *Binary* representa, como su propio nombre indica, el archivo binario a analizar. Esta clase tiene una relación con la clase *Function*, que representa las funciones. Esta clase está asociada a la clase *ControlFlowGraph*, que representa un CFG. Cada grafo está compuesto por uno o más bloques básicos, representados por la clase *BasicBlock*. Esta clase tiene como atributos la dirección del punto de entrada, la secuencia de instrucciones de cada bloque y los bytes del mismo. Por último, la clase *CFGHashNode* representa el hash único que tiene el CFG de cada función. Esta clase tiene una relación de uso

con la clase *HNSW*, que representa la estructura *HNSW* creada y que se explica con más detalle en la sección 3.3.2.

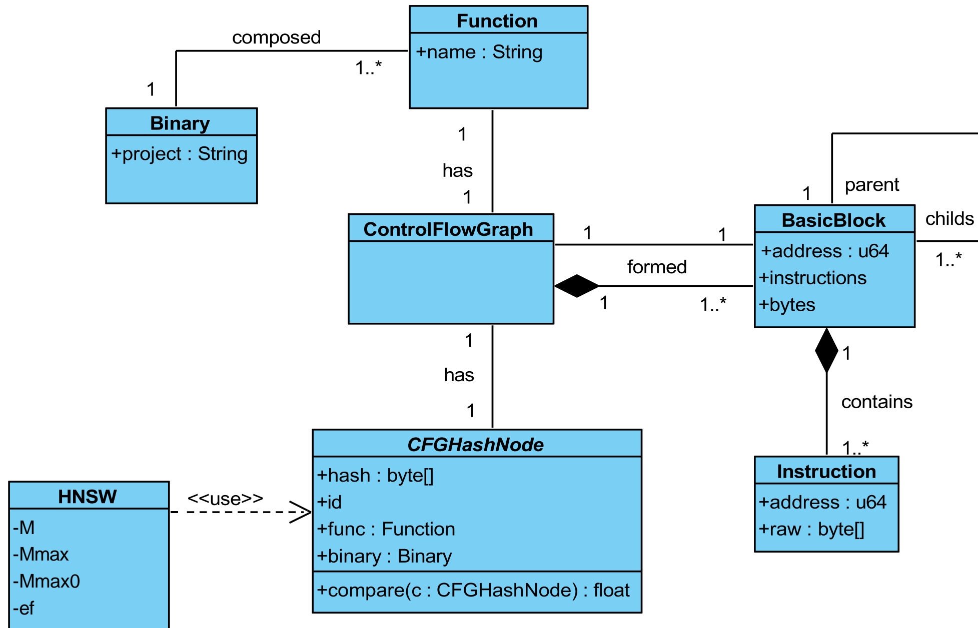

Figura 3.1: Diagrama de clases UML.

3.2.2. Diagrama de secuencia UML

Un diagrama de secuencia UML define la secuencia de interacciones de objetos del sistema entre sí. En este caso, se muestra el diagrama de secuencia del caso de uso obtener los hashes *TLSH* de un binario y añadirlos a la base de datos (veáse Figura 3.2).

Primero, se obtienen todas las funciones de la aplicación dada. Después, se itera en cada función para obtener su CFG, extrayendo sus bloques básicos y calculando su hash. El hash se construye enlazando los bytes de los bloques básicos y calculando el hash *TLSH* de los bytes obtenidos. Los bloques básicos se iteran en el orden tal y como los proporciona *angr*, que permite obtener los bloques básicos de una misma función en el mismo orden en cada ejecución. El hash *TLSH* calculado es el que se devuelve a la aplicación, que los va almacenando para finalmente llamar a *updateDB()* y actualizar así la base de datos.

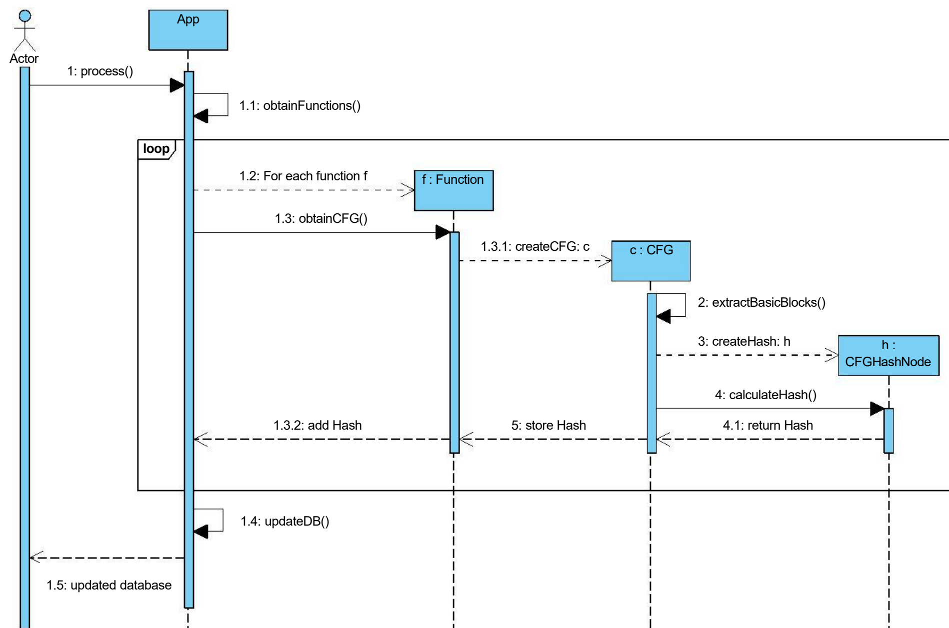

Figura 3.2: Diagrama de secuencia UML del caso de uso: obtener los hashes de un binario.

3.2.3. Diagrama de flujo UML

Un diagrama de flujo UML es la representación gráfica de un algoritmo o proceso. En este caso, la Figura 3.3 muestra el proceso que realiza la aplicación cuando el usuario utiliza la opción de añadir los hashes de las funciones de un binario dado a una base de datos.

En este diagrama se describe el proceso de entrada, comprobación y carga de los parámetros de entrada, se indica el momento de apertura o creación de la base de datos y procesamiento del binario, y finalmente la devolución de la base de datos actualizada.

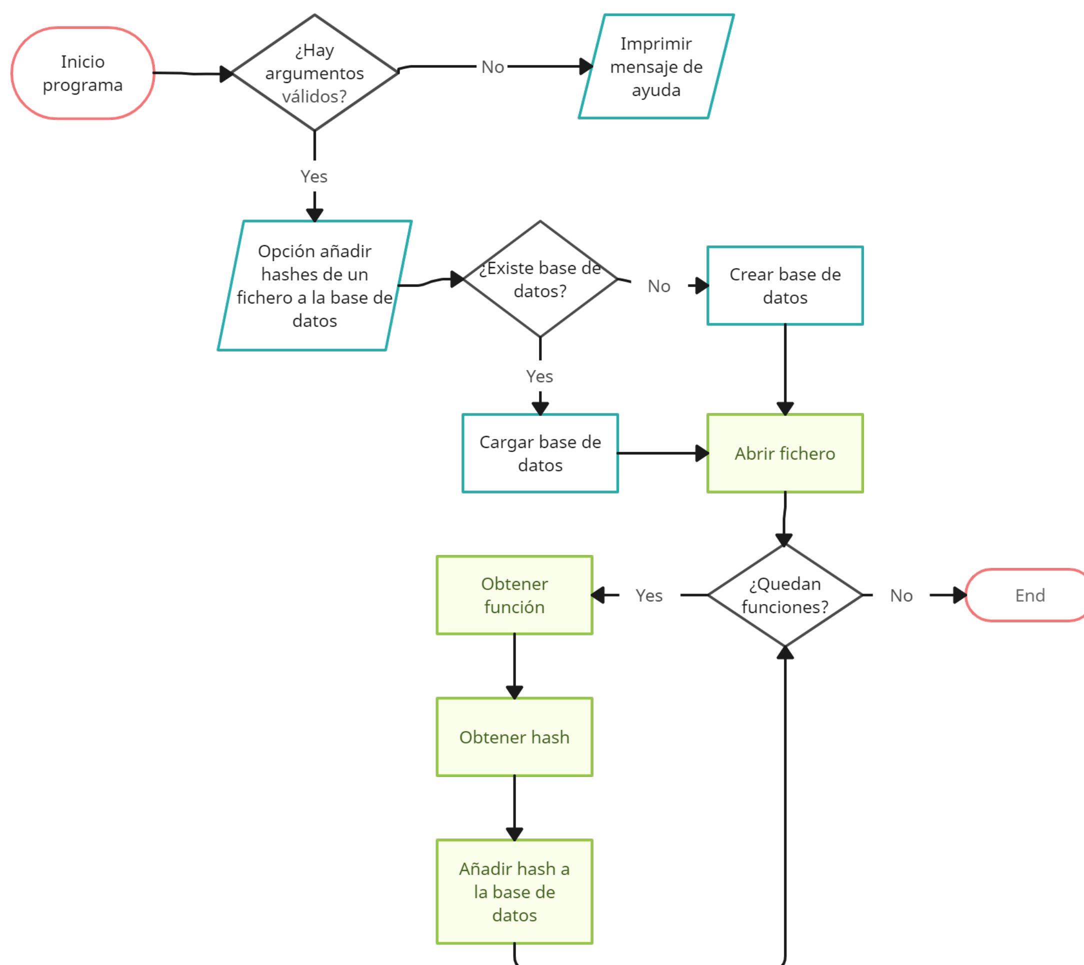

Figura 3.3: Diagrama de flujo UML con el proceso añadir hashes a la base de datos.

3.3. Funcionamiento y características

En esta sección se detallan las características y funcionamiento del código. En primer lugar, se resume la investigación realizada para la toma de decisiones. Posteriormente, se detallan las diferentes herramientas utilizadas durante el desarrollo del programa.

3.3.1. Investigación realizada

Para este trabajo, se ha hecho un estudio de las herramientas expuestas en la Tabla 3.1. En total son 7 herramientas de desensamblado de binarios. Estas herramientas tienen una alta popularidad y cubren casi todas las técnicas de desensamblado binario conocidas públicamente [28].

HERRAMIENTA	FECHA	LENGUAJE	FUENTE
Dyninst - 12.3.0	Febrero 2023	C/C++	GitHub [3]
Ghidra - 10.3	Mayo 2023	Java	GitHub [4]
angr - 9.2.48	Abril 2023	Python	GitHub [1]
Radare2 - 5.8.6	Mayo 2023	C	GitHub [7]
BAP - 2.5.0	Julio 2022	OCaml	GitHub [2]
Rev.ng	Abril 2023	C++	GitHub [8]
Jakstab - 0.10	Diciembre 2019	Java	GitHub [6]

Tabla 3.1: Herramientas analizadas

La generación de un CFG mediante herramientas de desensamblado binario consta de diferentes etapas [24], pero el estudio se ha centrado en los saltos indirectos, que son instrucciones que no apuntan directamente a una dirección de memoria, sino a una ubicación que contiene la dirección de destino; llamadas de cola *(tail calls)*, que sustituyen una instrucción call por un salto para aspectos relacionados con el marco de la pila y las funciones *non-returning*, que como su propio nombre indica son funciones no retornables.

La Tabla 3.2 muestra el resultado de la investigación realizada, agrupando por algoritmo y heurística utilizado por las herramientas para la creación del CFG. En esta tabla se dividen los algoritmos y heurísticas que usan las herramientas analizadas para determinar cuáles son las funciones no retornables, las llamadas de cola y los saltos indirectos.

Se puede concluir que hay algunos algoritmos/heurísticas que comparten todas o la mayoría de las herramientas estudiadas. Por ejemplo, la identificación de llamadas al sistema o funciones no retornables conocidas es una estrategia que utilizan todas las herramientas. También hay otros algoritmos que son únicos de una determinada herramienta, como por ejemplo determinar que el salto es una llamada de cola si el código entre el salto y su destino abarca varias funciones, que es propio de *Ghidra* [4].

	Algoritmos y heurísticas	Herramientas
Funciones no retornables	Identifica llamadas al sistema o funciones no retornables conocidas.	*Dyninst, Ghidra, angr, Radare2, BAP, Rev.ng, Jakstab*
	Identifica funciones sin instrucción ret como no retornables.	*Dyninst, Ghidra, angr, Radare2*
	Identifica funciones que terminan en una llamada a una función sin retorno.	*BAP*
Llamadas de cola	Trata los saltos a funciones simbólicas como llamadas de cola.	*Dyninst, angr*
	Trata los saltos a instrucciones como llamadas de cola si detecta que el marco de la pila se ha caído antes de esa instrucción.	*Dyninst, angr*
	Determina que un salto es una llamada de cola si el código entre el salto y su destino abarca varias funciones.	*Ghidra*
	Excluye los saltos condiciones de ser considerados como llamadas de cola.	*Ghidra, angr*
	Una llamada de cola no salta a la mitad de una función.	*angr*
	El objetivo de una llamada de cola no puede ser el objetivo de ningún salto condicional.	*angr*
	Identifica llamadas de cola basándose en si la distancia entre un salto y su destino excede cierto umbral.	*Radare2*
Saltos indirectos	Utiliza un análisis de conjuntos de valores simplificado.	*Dyninst, Ghidra, angr*
	Si el valor del índice tiene un rango mayor a un umbral, descarta la tabla de salto.	*Ghidra, angr, Radare2*
	Realiza un cierto tipo de análisis para determinar la tabla de saltos.	*Dyninst, Ghidra, angr, Rev.ng*
	Utiliza la propagación y el plegado de constantes para resolver saltos indirectos.	*Jakstab*
	Sigue patrones para determinar las tablas de salto.	*Dyninst, Ghidra, Radare2*

Tabla 3.2: Algoritmos y heurísticas utilizados por las herramientas estudiadas para la creación de CFGs

3.3.2. Tecnologías y herramientas utilizadas

angr

La primera decisión que se tomó para empezar a desarrollar de manera práctica el trabajo fue la determinación de elegir una herramienta de análisis binario. Teniendo en cuenta las herramientas mostradas en la Tabla 3.1 y el estudio realizado con ellas, se ha elegido *angr* [1] puesto que cumple con los requisitos establecidos.

angr [1] es una plataforma de análisis binario de código abierto construida en Python. Combina el análisis simbólico estático y dinámico, proporcionando herramientas para resolver una variedad de tareas. Entre las capacidades de *angr* se encuentra la recuperación de CFGs. La herramienta cuenta con una extensa API que facilita la construcción de nuevas herramientas usando sus capacidades de análisis. Mucha literatura reciente del campo de análisis de aplicaciones usa esta herramienta como base de sus trabajos, con lo que es una herramienta popular. Además, se actualiza muy a menudo. Otro detalle a tener en cuenta es que identifica funciones no retornables, llamadas de cola, llamadas indirectas y saltos indirectos para construir el CFG.

HNSW4Hashes

Para el desarrollo de la base de datos, así como para la comprobación de hashes dentro de esta, se ha utilizado la biblioteca *HNSW4Hashes* [21]. Esta biblioteca se ha obtenido de su repositorio oficial: `https://github.com/reverseame/HNSW4hashes`.

HNSW4Hashes es una implementación especializada de la estructura de datos *Hierarchical Navigable Small World* (HNSW) [19], adaptada para la búsqueda eficiente de coincidencias aproximadas de hashes. Concretamente, esta estructura permite realizar dos tipos de búsquedas [21]:

- **Basada en vecinos cercanos:** Dado un nodo de consulta, se devuelven aquellos k nodos más cercanos a este, siendo k un número dado como parámetro en la búsqueda. En este contexto, el término nodo hace referencia a un nodo de la estructura *HNSW* creada.
- **Basada en umbral:** Dado un nodo de consulta, se devuelven todos aquellos vecinos que superen cierto umbral de similitud dado como parámetro por el usuario.

La Figura 3.1 (introducida en la Sección 3.2) muestra cómo se relaciona la biblioteca con el sistema, concretamente con la clase *CFGHashNode*. Esta biblioteca se

usa en la herramienta desarrollada para almacenar la base de datos, que se guarda en un fichero externo. Para lograr el desarrollo correcto en el caso de que el usuario quiera añadir hashes a la base de datos, la aplicación comprueba si existe ya esta estructura. Si no es así, crea la estructura *HNSW* y la almacena en un archivo. Es entonces cuando realiza la actualización de la base de datos, añadiendo los nodos correspondientes. Si la estructura ya había sido creada, entonces simplemente realiza la inserción en la base de datos.

La estructura *HNSW* tiene los siguientes parámetros de configuración que determinan el comportamiento y desempeño de la estructura:

- *M*: Controla el nivel de conexión entre los nodos en la estructura. Es el número máximo de vecinos que tendrá un nodo en el momento de ser insertado.
- *ef*: Determina la cantidad de nodos vecinos que se explorarán durante la búsqueda.
- *Mmax*: Establece el número máximo de vecinos que un nodo puede tener en todos los niveles de la estructura, excepto en la capa cero.
- $Mmax_0$: Establece el número máximo de vecinos que un nodo puede tener en la capa cero.

En la implementación del código esta estructura está parametrizada de la siguiente manera: *M=64, ef=64, Mmax=128, Mmax0=256*. Se ha elegido esta configuración puesto que muestra un rendimiento óptimo en términos de precisión y proporciona una notable mejora en los tiempos de ambos métodos de búsqueda soportados, según [21].

3.4. Limitaciones

La principal limitación del trabajo es que la herramienta se encuentra fuertemente acoplada a *angr*. También existen otras limitaciones, como la dependencia de herramientas externas para su correcto funcionamiento, cuya configuración e instalación puede ser complicada. Otra limitación es la falta de pruebas con excesivos volúmenes de datos. Además, realizando las pruebas se ha tenido que modificar el límite de recursividad en Python para evitar errores durante la ejecución del programa. Concretamente, el error sucedía dentro de la biblioteca *HNSW* cuando había un gran número de funciones que añadir a la base de datos mediante la biblioteca *pickle*.

Otra de las flaquezas es a la hora de generar los hashes *TLSH*, puesto que si la función tiene menos de 50 bytes no genera un hash válido y esto hace que se pierdan funciones. Siguiendo con las limitaciones que tienen los hashes *TLSH*, hay que comentar que no son 100 % precisos y pueden generar falsos positivos. La comparación basada en *TLSH* es eficiente para identificar similitudes estructurales, pero no considera la semántica del código. Esto significa que dos bloques de datos que son estructuralmente similares, pero tienen significados diferentes, podrían generar hashes similares. Esto puede llevar a falsos positivos o a la falta de detección de diferencias importantes en el contenido.

Por último, existe otra limitación respecto a los datos de entrada. Actualmente, admite todo tipo de datos sin realizar validaciones de ningún tipo por parte del sistema. Es decir, no comprueba que el fichero de entrada sea un ejecutable, pudiendo el usuario introducir otro tipo de archivo.

3.5. Disponiblidad del código

Todo el código fuente utilizado para el desarrollo del sistema se encuentra en un repositorio público de GitHub (Repositorio-binarycomp4hashes). El código se encuentra bajo la licencia *GNU/GPLv3*.

Capítulo 4

Evaluación y discusión de resultados

Con el propósito de verificar el sistema creado, en este capítulo se evalúa su rendimiento y se valida su funcionamiento adecuado. Para alcanzar este objetivo, se realizan una serie de pruebas y experimentos que posibilitan la obtención de métricas cuantitativas y la realización de las comparaciones pertinentes.

En este capítulo se comenta en primer lugar el entorno de experimentación y se explica el conjunto de datos usado. Finalmente, se realiza un análisis de los resultados obtenidos y se discute el rendimiento del sistema.

4.1. Evaluación

La evaluación consiste en determinar cuántas funciones similares encuentra el programa en los binarios de `binutils` [5] si se comparan con el binario `strings`, de la misma colección de herramientas. Para realizar esta prueba lo primero es compilar todo el conjunto de datos con gcc (sin compilación cruzada). Luego, se siguen los siguientes puntos:

1. Para cada binario del conjunto de pruebas se obtienen las funciones que comparte con el binario `strings` mediante línea de comandos. Para ello, se usan los comandos `nm` y `comm`. Estos datos serán considerados como "verdad absoluta".
2. Se utiliza la herramienta desarrollada y se insertan en la base de datos los hashes de las funciones del binario `strings`.
3. Utilizando de nuevo la herramienta, para cada binario se obtienen las funciones similares que encuentra mediante búsqueda basada en vecinos y búsqueda basada en umbral.

4. Se realiza una comparación de los resultados obtenidos.

4.1.1. Definición del entorno

El entorno seleccionado para realizar las pruebas consiste en un equipo personal con un procesador Intel(R) Core(TM) i5-10300H a 2.50GHz de 8 núcleos y 16 hilos. El sistema cuenta con 8GiB de RAM DDR4 a 3200MHz y ejecuta un sistema operativo Ubuntu en su versión 23.04.

4.1.2. Conjunto de datos para pruebas

El conjunto de datos que se va a utilizar para las pruebas es la colección de herramientas `binutils` [5] obtenida del siguiente repositorio: `https://github.com/andrew-d/static-binaries/tree/master/binutils`. Este repositorio contiene numerosos binarios enlazados estáticamente, junto con los Dockerfiles y otros scripts de compilación que se pueden utilizar para construirlos.

Antes de comenzar las pruebas se va a compilar `binutils` sin compilación cruzada, sin ningún tipo de optimización y sin quitar los símbolos. Hay que tener en cuenta que las pruebas se realizan con la versión de `binutils` 2.25 y con la versión de gcc 11.4.0.

4.2. Discusión de resultados

4.2.1. Número de funciones encontradas en los binarios

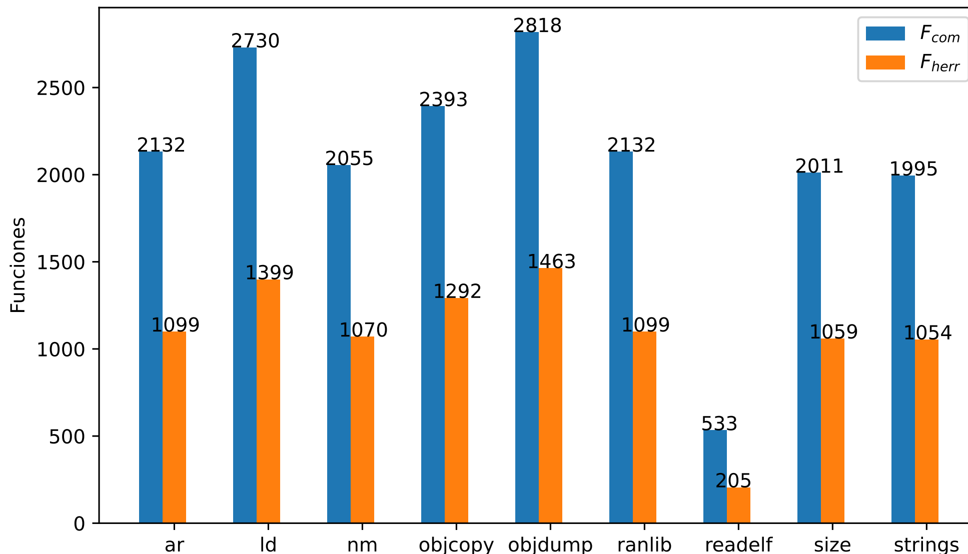

Figura 4.1: Número de funciones totales obtenidas por comandos (F_{com}) comparadas con el número de funciones totales obtenidas con la herramienta desarrollada (F_{herr})

La Figura 4.1 muestra el número de funciones totales obtenidas por el comando nm (F_{com}) y el número de funciones totales obtenidas con la herramienta desarrollada (F_{herr}). En todos los binarios evaluados se puede comprobar que F_{herr} es menor que F_{com}. Una posible explicación a esto es que las funciones que cuenta la herramienta son únicamente aquellas que se insertan en la base de datos. Esta base de datos no inserta las funciones que no tienen hash *TLSH*, que ocurre cuando la función tiene menos de 50 bytes, lo que conlleva a una pérdida de funciones. Por otro lado, el comando nm, usado para contabilizar las funciones mediante línea de comandos, se utiliza para mostrar información sobre los símbolos presentes en archivos binarios. Estos símbolos son elementos como variables globales, funciones y otras entidades definidas en el código de un programa. El resultado de este comando se utiliza en su totalidad, sin diferenciar si el símbolo corresponde a una función o no. Por lo tanto, se puede haber contabilizado como función algo que realmente no lo es.

4.2.2. Número de funciones similares encontradas en los binarios mediante búsqueda basada en vecinos

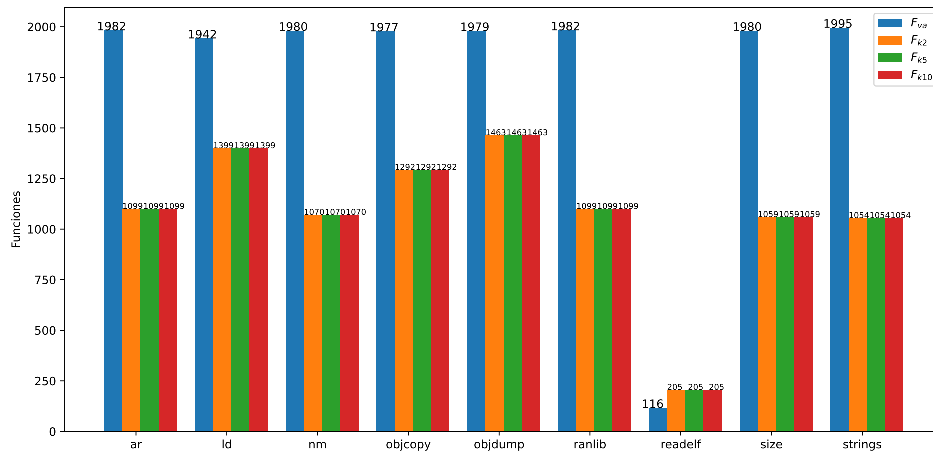

Figura 4.2: Número de funciones "verdad absoluta" (F_{va}) comparadas con el número de funciones similares obtenidas con búsqueda basada en vecinos (F_{k2}, F_{k5}, F_{k10})

La Figura 4.2 compara el número de funciones "verdad absoluta" (F_{va}) con el número de funciones similares que encuentra la aplicación realizando búsqueda basada en vecinos cercanos, siendo este número de vecinos 2, 5 o 10 (F_{k2}, F_{k5}, F_{k10}, respectivamente). En esta figura destaca el caso del binario *readelf*, en el que F_{va} es menor que F_{k2}, F_{k5} o F_{k10}, cuando en todos los demás binarios sucede al revés. Esto es debido a que los valores de F_{k2}, F_{k5} y F_{k10} se basan en la búsqueda de funciones similares, mientras que los datos de "verdad absoluta" muestran el número de funciones iguales. Teniendo esto en cuenta, es lógico pensar que el número de funciones de F_{k2}, F_{k5} y F_{k10} debe ser igual o mayor que el de F_{va}. Sin embargo, esto no es así ya que la base de búsqueda no es la misma (veáse Figura 4.1). Además, el número de F_{k2}, F_{k5} y F_{k10} se corresponde exactamente con el número de funciones totales obtenidas con la herramienta desarrollada (F_{herr}).

Para realizar la búsqueda basada en vecinos cercanos se define *k* como parámetro en la búsqueda, siendo *k* el número de nodos más cercanos a devolver al nodo de consulta. En este tipo de búsqueda también se define *ef*, que determina la cantidad de nodos vecinos que se exploran durante la búsqueda. Este parámetro *ef* no hay que confundirlo con el parámetro del mismo nombre en la creación de la estructura *HNSW*, puesto que ese valor es invariable durante la ejecución del programa. Las pruebas realizadas en la Figura 4.2 tienen como valor *ef=1*. Observando que los

resultados obtenidos con todos los valores de *k* son los mismos, se ha procedido a realizar las mismas pruebas pero cambiando el valor de *ef* a 4.

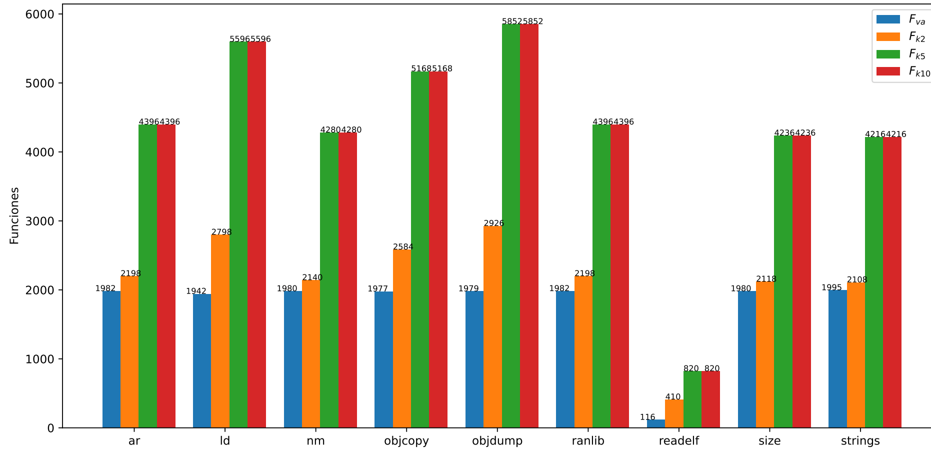

Figura 4.3: Número de funciones "verdad absoluta" (F_{va}) comparadas con el número de funciones similares obtenidas con búsqueda basada en vecinos (F_{k2}, F_{k5}, F_{k10}) (con $ef = 4$)

En la Figura 4.3 se observan los resultados obtenidos del número de funciones "verdad absoluta" (F_{va}) con el número de funciones similares obtenidas con búsqueda basada en vecinos (F_{k2}, F_{k5}, F_{k10}) con la configuración *ef=4*. Se observa que para los valores F_{k5} y F_{k10} el número de funciones similares encontradas son las mismas. Esto se debe a que el valor de *ef* es inferior al valor de *k* en ambos casos, por lo que la búsqueda solamente devuelve 4 nodos como máximo. En consecuencia, el número de funciones similares encontradas basadas en la búsqueda con vecinos es mayor al número de funciones "verdad absoluta" en todos los binarios.

4.2.3. Número de funciones similares encontradas en los binarios mediante búsqueda basada en umbral

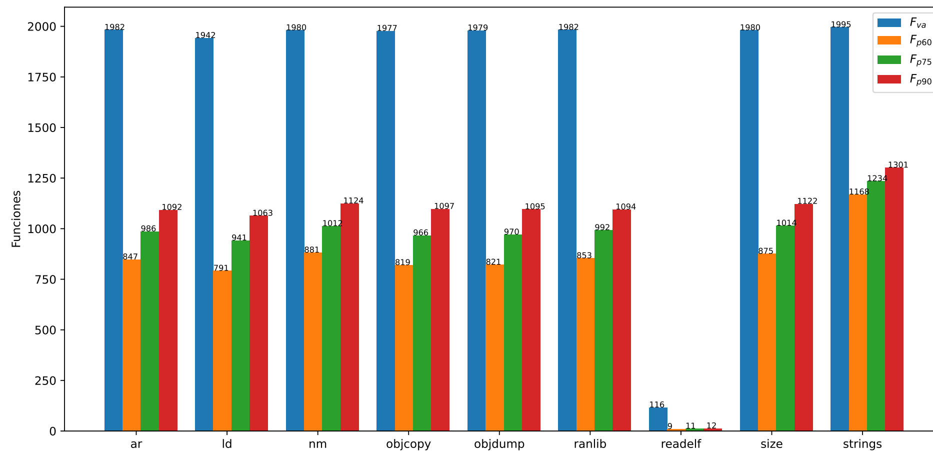

Figura 4.4: Número de funciones "verdad absoluta" (F_{va}) comparadas con el número de funciones similares obtenidas con búsqueda basada en umbral (F_{p60}, F_{p75}, F_{p90})

La Figura 4.4 compara el número de funciones "verdad absoluta" (F_{va}) con el número de funciones similares que encuentra la aplicación realizando búsqueda basada en umbral, siendo este umbral 60, 75 o 90 (F_{p60}, F_{p75}, F_{p90}). En todos los binarios evaluados el número de funciones similares encontradas aumenta según se va aumentando el valor del umbral en la búsqueda. Esto es así porque en *TLSH*, una similitud de 0 entre dos elementos significa la máxima coincidencia entre ellos. Por lo tanto, un valor de umbral de similitud 60 indica a la herramienta que busque funciones con mayor similitud que el valor de 90. Por otro lado, el valor de F_{va} es mayor que el valor de F_{pX} en todos los casos, siendo X el valor del umbral en la búsqueda. F_{va} es el número de funciones iguales que se obtienen por línea de comandos.

Capítulo 5

Conclusiones, problemas encontrados y trabajo futuro

Este capítulo presenta las conclusiones del trabajo realizado, así como los problemas y limitaciones encontrados en el desarrollo del proyecto y las posibles líneas de investigación en el futuro.

5.1. Conclusiones

En este trabajo se ha enfrentado el reto de obtener una herramienta capaz de almacenar y comparar hashes de distintos binarios para mostrar la similitud entre ellos. Los resultados del trabajo permiten concluir que la utilización de *TLSH* para calcular hashes de funciones basadas en bloques básicos del CFG es una técnica eficiente para detectar similitudes entre binarios. Sin embargo, es importante considerar la escalabilidad y la mantenibilidad de la aplicación puesto que son factores críticos. En este caso, a medida que se agregan más binarios la base de datos crece y se incrementan los tiempos de inserción y consulta, con lo que es un aspecto a considerar.

Resumiendo, se ha logrado obtener una aplicación que cumple el objetivo establecido y demuestra cómo la comparación de binarios mediante el hash *TLSH* de sus funciones obtenidos a través de los bloques básicos del CFG es útil, sentando las bases para investigaciones futuras.

5.2. Problemas encontrados

Llevar a cabo este proyecto ha brindado la posibilidad de adquirir un profundo entendimiento sobre el ámbito de los algoritmos de similitud aproximada y CFGs, así como de la comparación de binarios. Dado que el trabajo involucraba conceptos relativamente nuevos, se han enfrentado ciertas complicaciones que han sido superadas con éxito a lo largo del desarrollo del proyecto gracias a la existencia de artículos y recursos.

A la hora de la implementación es cuando más problemas se han encontrado durante el desarrollo de este proyecto. En un principio, se planteó el uso de la herramienta *Ghidra* [4], pero se descartó ya que era demasiado complejo trabajar con ella, además de obtener un sistema más acoplado y todavía con más dependencias.

Finalmente, en la etapa de pruebas se tuvieron muchos inconvenientes. Uno de estos problemas está relacionado con las dependencias del proyecto, que derivó en la necesidad de actualizar el sistema operativo utilizado en las pruebas y ciertas dependencias de software.

5.3. Trabajo futuro

El desarrollo de este proyecto es susceptible a líneas de investigación en el futuro. A continuación se van a detallar algunas de ellas:

- Desarrollar un sistema no tan acoplado a *angr* [1], la herramienta utilizada en el análisis de binarios del sistema.
- Optimizar el código para mejorar su rendimiento, así como mejorar la programación de la herramienta introduciendo robustez.
- Aportar a la herramienta más valor mediante nuevas funcionalidades, como por ejemplo, devolver el CFG en una imagen o extraer para cada función estudios de complejidad y entropía.
- Ampliar la batería de pruebas expuestas mediante el uso de la herramienta en otros conjuntos de datos y la evaluación experimental de la herramienta.
- Realizar una comparativa de la herramienta desarrollada en este trabajo con otras herramientas de búsqueda de similitudes en binarios.

Bibliografía

[1] angr. [Online:] https://github.com/angr/angr. Accedido el 4 de septiembre, 2023.

[2] BAP: Binary Analysis Platform. [Online:] https://github.com/BinaryAnalysisPlatform/bap. Accedido el 4 de septiembre, 2023.

[3] Dyninst. [Online:] https://github.com/dyninst/dyninst. Accedido el 4 de septiembre, 2023.

[4] Ghidra Software Reverse Engineering Framework. [Online:] https://github.com/NationalSecurityAgency/ghidra. Accedido el 4 de septiembre, 2023.

[5] GNU Binutils. [Online:] https://www.gnu.org/software/binutils/. Accedido el 4 de septiembre, 2023.

[6] Jakstab. [Online:] https://github.com/jkinder/jakstab/. Accedido el 4 de septiembre, 2023.

[7] Radare2: Libre Reversing Framework for Unix Geeks. [Online:] https://github.com/radareorg/radare2. Accedido el 4 de septiembre, 2023.

[8] Rev.ng. [Online:] https://github.com/revng/revng. Accedido el 4 de septiembre, 2023.

[9] Frances E Allen. Control Flow Analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[10] Frank Breitinger. *Approximate Matching: Definition and Terminology*. US Department of Commerce, National Institute of Standards and Technology, 2014.

[11] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting Self-mutating Malware Using Control-Flow Graph Matching. In *Detection of Intrusions and Malware & Vulnerability Assessment: Third International Confe-*

rence, DIMVA 2006, Berlin, Germany, July 13-14, 2006. Proceedings 3, pages 129–143. Springer, 2006.

[12] Silvio Cesare, Yang Xiang, and Wanlei Zhou. Control Flow-Based Malware VariantDetection. *IEEE Transactions on Dependable and Secure Computing*, 11(4):307–317, 2013.

[13] Sebastian Eschweiler, Khaled Yakdan, Elmar Gerhards-Padilla, et al. discovRE: E cient Cross-Architecture Identification of Bugs in Binary Code. In *Ndss*, volume 52, pages 58–79, 2016.

[14] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. Extracting Conditional Formulas for Cross-Platform Bug Search. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 346–359, 2017.

[15] Víctor Gayoso Martínez, Fernando Hernández Álvarez, and Luis Hernández Encinas. State of the Art in Similarity Preserving Hashing Functions. 2014.

[16] Amanda Lee and Travis Atkison. A Comparison of Fuzzy Hashes: Evaluation, Guidelines, and Future Suggestions. In *Proceedings of the SouthEast Conference*, pages 18–25, 2017.

[17] Yuping Li, Jiyong Jang, and Xinming Ou. Topology-Aware Hashing for Effective Control Flow Graph Similarity Analysis. In *Security and Privacy in Communication Networks: 15th EAI International Conference, SecureComm 2019, Orlando, FL, USA, October 23-25, 2019, Proceedings, Part I 15*, pages 278–298. Springer, 2019.

[18] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 389–400, 2014.

[19] Yu A Malkov and Dmitry A Yashunin. E cient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[20] Miguel Martín-Pérez, Ricardo J. Rodríguez, and Frank Breitinger. Bringing order to approximate matching: Classification and attacks on similarity digest

algorithms. *Forensic Science International: Digital Investigation*, 36:301120, 2021. DFRWS 2021 EU - Selected Papers and Extended Abstracts of the Eighth Annual DFRWS Europe Conference.

[21] Daniel Huici Meseguer. Búsqueda eficiente de hashes de similitud aproximada. Master's thesis, 2023.

[22] Vitor Hugo Galhardo Moia and Marco Aurélio Amaral Henriques. Similarity Digest Search: A Survey and Comparative Analysis of Strategies to Perform Known File Filtering Using Approximate Matching. *Security and Communication Networks*, 2017:1306802, Sep 2017.

[23] Jonathan Oliver, Chun Cheng, and Yanggui Chen. TLSH - A Locality Sensitive Hash. In *2013 Fourth Cybercrime and Trustworthy Computing Workshop*, pages 7–13. IEEE, 2013.

[24] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE symposium on security and privacy (SP)*, pages 833–851. IEEE, 2021.

[25] Miguel Martín Pérez. *E ectiveness of Similarity Digest Algorithms for Binary Code Similarity in Memory Forensic Analysis*. PhD thesis, Universidad de Zaragoza, 2022.

[26] OMG Available Specification. OMG Unified Modeling Language (OMG UML),Superstructure, V2.1.2. *Object Management Group*, 70, 2007.

[27] Fish Wang and Yan Shoshitaishvili. Angr - The Next Generation of Binary Analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

[28] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. From Hack to Elaborate Technique—A Survey on Binary Rewriting. *ACM Computing Surveys (CSUR)*, 52(3):1–37, 2019.

Anexo A

Seguimiento de proyecto fin de máster

En este apéndice se expone la estructuración del proyecto, el número de horas totales y el diagrama de Gantt.

Las fases en las que se ha dividido el desarrollo del TFM han sido las siguientes:

- Estudio de las herramientas de análisis binario mediante CFG: la finalidad es obtener nociones básicas sobre las herramientas y el CFG, eligiendo una de ellas.
- Estudio de los principales algoritmos de similitud aproximada: estas técnicas son usadas para encontrar elementos similares en grandes conjuntos de datos.
- Desarrollo e implementación del programa: la finalidad es desarrollar una nueva herramienta para encontrar las funciones similares (dentro de una base de datos) a un binario dado por el usuario.
- Observación y estudio de los resultados dados en los experimentos realizados: el desarrollo y pruebas se han hecho con Python.

El número de horas totales ha sido de 423, cuyo desglose se puede ver en la Tabla A.1. El número de horas trabajadas en la etapa de investigación ha sido de 75. Esta etapa ha consistido en establecer el marco teórico sobre el que se va a basar el proyecto, así como la investigación y elección de las tecnologías usadas para el mismo. En la fase de diseño se han invertido un total de 48 horas y el objetivo ha sido definir los requisitos tanto funcionales como no funcionales del sistema y crear los diagramas pertinentes. La etapa de desarrollo ha sido la más costosa, con un total de

120 horas. En ella se engloba todo lo que tiene relación con la implementación de la aplicación. El período de evaluación ha consistido en 93 horas. Estas horas han ido repartidas entre elegir el conjunto de datos para las pruebas, el diseño de las mismas y el desarrollo e implementación y comprobación de los resultados obtenidos. Por último, la etapa de documentación, donde se han invertido 87 horas.

Actividad	Nº horas
Investigación	75
Diseño	48
Desarrollo	120
Evaluación	93
Documentación	87
Horas totales	423

Tabla A.1: Número de horas trabajadas

Adicionalmente, en la Figura A.1 se muestra el diagrama de Gantt del proyecto.

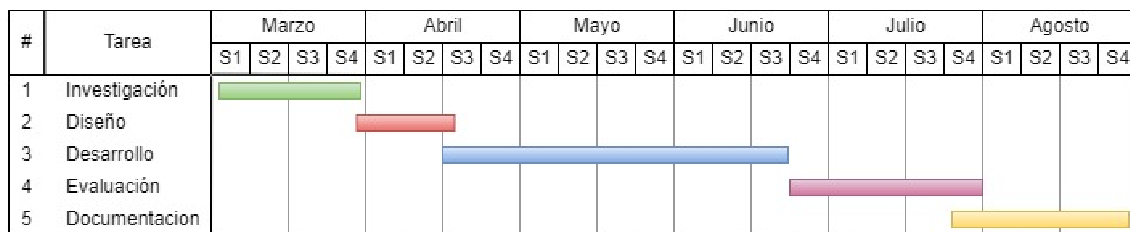

Figura A.1: Diagrama de Gantt.