Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Tecnología de la Informacíon

Generación de Diagramas de Flujo a partir de código objeto en ELF

Autor: Mario López Cuesta

Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en Tecnología de la Informacíon

Generación de Diagramas de Flujo a partir de código objeto en ELF

Autor: Mario López Cuesta

Tutor: Valentín Cardeñoso Payo

Agradecimientos

Quiero agradecer a toda mi familia y en especial a mis padres por el apoyo, paciencia y comprensión durante todos mis años de educación. A mi pareja que me ha apoyado y animado siempre a continuar. A todos mis amigos de la escuela con los que he pasado tantos ratos y siempre estaban dispuestos a echar una mano. A mi tutor Valentín Cardeñoso que me ha apoyado, ayudado y aconsejado desde que plantee la idea del proyecto hasta su finalización. Por lo ultimo, gracias a todos los profesores que me han formado durante mi etapa académica en la escuela de Ingeniería Informática.

Resumen

Este trabajo de fin de grado presenta el desarrollo de un aplicación de terminal destinada a procesar código objeto en formato ELF con la finalidad de transformarlo en una representación abstracta para ayudar a su entendimiento. Para la representación del fichero ELF se utilizarán gráficos de control de flujo del fichero ejecutable. Para facilitar la representación del código se ha analizado la estructura y composición de el formato ELF.

Para el desarrollo del programa se ha utilizado el lenguaje de programación RUST, se ha escogido este lenguaje debido a que es un lenguaje relativamente nuevo, desarrollado en gran parte gracias a la comunidad y proporciona un enfoque multi-paradigma (programación funcional pura, por procedimientos, imperativa…) además de definirse como seguro y eficiente.

Abstract

This end-of-degree project presents the development of a terminal application intended to process object code in ELF format in order to transform it into an abstract representation to aid understanding. For the representation of the ELF file, flow control charts of the executable file will be used. To facilitate the representation of the code, the structure and composition of the ELF format have been analyzed.

The RUST programming language has been used to develop the program, this language has been chosen because it is a relatively new language developed by the community and provides a multiparadigm approach (pure functional programming, procedural, imperative …) in addition to being defined as safe and efficient programming language

Índice general

Índice de cuadros

Índice de figuras

Parte I

Objeto, Concepto y Método

Capítulo 1

Introducción

Quedan ya muy lejos los tiempos en que las máquinas se programaban en lenguaje máquina o, incluso, en lenguaje ensamblador. En la actualidad, hay una inmensa variedad de lenguajes de programación de alto nivel que requieren del uso de compiladores o intérpretes que se encarguen de trasladas a lenguaje máquina el código expresado en un lenguaje más cercano al que los programadores usan para comprender y formular sus soluciones. El proceso de compilación incluye, habitualmente, fases de optimización y generación de código que hacen que el código objeto final resultante se ajuste a unos flujos de ejecución del código que no coinciden exactamente con los que se expresan en el código fuente.

En el ámbito académico como a nivel profesional, tanto desarrolladores como analistas están interesados y buscan soluciones para saber que es lo que de verdad se esta ejecutando en el procesador, ya sea para optimizar y hacer mas seguro el código propio, para comprender mejor el código de terceros o para analizar posibles piezas de malware, sin tener que leer código maquina directamente del fichero resultado de una compilación. Estos ficheros contienen lenguaje ensamblador que es un lenguaje intermedio difícil de interpretar para los humanos. De esta carencia surge la necesidad de las interpretaciones abstractas del código.

Partiendo de ficheros que contienen código objeto de programas en formato ELF con arquitectura de 64 bits, se quiere elaborar el grafo de ejecución o flujo asociado a cada programa de forma automática. Para ello se quiere diseñar, construir y probar una herramienta que permita a través del análisis estático del fichero producirá el desensamblado de código objeto ELF con el que se generara una representación abstracta de la misma basada en grafos de los posibles flujos de ejecución que el programa puede tener. Solo será capaz de mostrar los posibles caminos, ya que flujo de ejecución solo es mostrado en tiempo de ejecución y depende de la entrada que el programa reciba. Esta herramienta facilitará el análisis de programas, tanto a nivel de detección de errores o fragmentos optimízales, cuando no se disponga del código fuente.

1.1 Motivación

El código fuente del software que usamos a diario no siempre está disponible. Un desensamblador como IDA Pro es capaz de crear mapas de su ejecución para mostrar las instrucciones binarias que realmente ejecuta el procesador en una representación simbólica llamada lenguaje ensamblador. Este proceso de desmontaje permite a los especialistas en software analizar los programas que se sospecha son de naturaleza nefasta, como el software espía o el malware. Sin embargo, el lenguaje ensamblador es difícil de leer y entender. Es por eso por lo que se han implementado técnicas avanzadas en IDA Pro para hacer que ese código complejo sea más legible.

En algunos casos, es posible revertir el programa binario, a un nivel bastante cercano, al código fuente original que lo produjo. El mapa del código del programa se puede procesar posteriormente para una investigación más detallada.

1.2 Estructura de la memoria

La memoria ha sido estructurada en tres partes divididas en capítulos, que estos entre las tres partes hacen un total de 10 capítulos, una sección de anexos y para concluir se encuentra la sección de bibliográfica.

La Parte I denominada *Objetivo, Concepto y Método* con los capítulos 1 *Introducción*, 2 *Objetivos y Alcance* y 3 *Metodología*. En esta parte se presenta la idea del proyecto en general, describiendo la motivación, los objetivos a cumplir en el proyecto y el alcance que tendrá, también se describen los plazos en los que va a ser desarrollada, enunciando las distintas etapas y entregables que tendrá, y la metodología que va a ser seguida en la gestión del proyecto.

La Parte II denominada *Marco Conceptual y Contexto* consta solo de dos capítulos 4 *Marco Conceptual* y 5 *Soluciones Existentes*. En esta segunda parte se pone en contexto la idea del proyecto, describiendo los temas que van a ser tratado y conceptos teóricos que influyen en el desarrollo del proyecto. También se describen herramientas y proyectos similares o que tratan los mismos temas y que ya están disponibles para el publico general.

La Parte III denominada *Desarrollo del Sistema* con los capítulos 6 *Análisis*, 7 *Diseño*, 8 *Implementación*, 9 *Pruebas*, 10 *Conclusiones*. Esta es la parte mas extensa y es la que describe desde el principio hasta el final el proceso de creación de la herramienta.

La sección de anexos cuenta con la descripción de algunas estructuras de código usadas, manuales de uso e instalación y cierto código utilizado para la realización de la memoria que no esta en el proyecto final o que es interesante de describir en la memoria.

Capítulo 2

Objetivos y Alcance

2.1 Objetivos

En este capítulo se va a tratar los objetivos definidos para el TFG y especificar el trabajo necesario a realizar,constituirá la sección alcance del proyecto.

2.1.1 Objetivo

El objetivo principal de este TFG es el desarrollo de una aplicación de terminal para sistemas GNU/Linux encargada de procesar ficheros binarios tipo ELF y generar a partir de ellos el gráfico de control de flujo de ejecución del fichero.

Siendo este el objetivo final, para llegar a conseguirlo se definen los siguientes objetivos:

- Estudiar y analizar el funcionamiento de las herramientas existentes similares.
- Diseñar e implementar herramienta. Programa encargado de procesado del fichero y la generación de la representación gráfica.
- Diseñar plan de pruebas para determinar la validez de la herramienta.
- Comparar resultados con los de herramientas similares.
- Realizar manual de uso e instalación de la herramienta.
- Realizar estudio de la estructura del fichero ELF e identificar partes necesarias para la realización del proyecto.
- Búsqueda de información de posibles formatos de salida y herramientas para representar grafos.

2.2 Alcance

Con el fin de definir el alcance del proyecto es necesario especificar el trabajo que ha de ser realizado en el proyecto. Pera definir el trabajo necesario para llevar a cabo los objetivos redactados en la Sección 2 se han identificado las siguientes fases las cuales se han dividido en tareas a realizar.

1. Análisis y descripción

1.1. Planteamiento de la idea principal
1.2. Visión general del proyecto y selección de las tecnologías a usar.
1.3. Búsqueda de soluciones similares existentes
1.4. Planificación de fases, iteraciones y riesgos.
1.5. Familiarización con tecnologías y herramientas.

2. Planteamiento y descripción de los entregables

2.1. Licitación de requisitos
2.2. Búsqueda y descripción de casos de uso
2.3. Modelo de dominio análisis
2.4. Planteamiento y diseño de la arquitectura
2.5. Modelo dominio diseño
2.6. Diagrama de secuencia de casos de uso críticos

3. Desarrollo de entregables

3.1. Apertura y análisis del fichero
3.2. Desensamblado a partir de código crudo
3.3. División de las instrucciones en bloques básicos
3.4. Construcción de grafo a partir de bloques básicos
3.5. Construcción de gráfico visual a partir del grafo

4. Revisión

4.1. Prueba ante stackholders
4.2. Análisis de posibles mejoras y correcciones
4.3. Implementación de correcciones y mejoras
4.4. Elaboración de manual de usuario

Capítulo 3

Metodología

Todo trabajo requiere un método y para este TFG se va a utilizar la metodología agile, en concreto SCRUM. El uso de SCRUM implica la asignación de los roles dentro de las personas encargadas dentro del proyecto, debido a que es un proyecto individual una sola persona ejercerá todos los roles. Y las reuniones periódicas serán las realizadas con el profesor responsable del TFG. Junto con SCRUM se va a utilizar el desarrollo incremental iterativo, en el cual en cada iteración se ira añadiendo funcionalidades y documentación al proyecto.

Las fases van a ser divididas en “pequeños proyectos”llevados a cabo en las iteraciones, los cuales tendrán las etapas: planificación, desarrollo, pruebas y documentación. Estos proyectos ampliarán la funcionalidad y corregirán errores de versión anteriores.

3.1 Fases y costes

El desarrollo del TFG se ha dividido en 4 fases. En cada fase se elaborarán distintas partes del proyecto. Las fases su duración el tema que será tratado esta descrito en los Cuadros 3.1, 3.2, 3.3 y 3.4.

Fase 01: Análisis y descripción inicial del proyecto
Duración: 1 semana
En esta fase se expondrá el planteamiento de la idea principal, junto con la visión general del proyecto, se escogerán las tecnologías a utilizar. También se buscará informático sobre el contexto y se familiarizará con las herramientas y tecnologías a usar.

Cuadro 3.1: Fase 01

Fase 02: Planteamiento y descripción de los entregables
Duración: 2 semana
En esta fase se realizará el análisis del proyecto. Se realizará la investigación y especificación del problema y la definición de componentes.

Cuadro 3.2: Fase 02

Fase 03: Desarrollo de entregables
Duración: 3 semana
En esta fase se realizará el desarrollo del software basándose en la descripción realizada en la fase de análisis del proyecto junto con las pruebas.

Cuadro 3.3: Fase 03

Fase 04: Revisión
Duración: 1 semana
Revisión de la memoria y verificación de la herramienta desarrollada. También se incluyen solucionar los posibles errores contenidos en el código y el desarrollo del manual de uso.

Cuadro 3.4: Fase 04

3.2 Gestión de Riesgos

En el desarrollo de cualquier proyecto pueden surgir contratiempos que retrasen o modifiquen el proyecto. Para evitar que el efecto de estos en el proyecto sea grave y disminuir su impacto se realiza un análisis previo de los obstáculos que pueden ser encontrado y su posible impacto en el proyecto. Junto con cada riesgo se describen los planes de reducción de impacto y de contingencia.

En las tablas 3.5 a la 3.13 se describen los riesgos que se han tenido en cuenta a la hora de desarrollar el proyecto. En la Figura 3.1 se muestra la relación entre la probabilidad de ocurrencia y las consecuencias sobre el proyecto.

		Probabilidad				
		Raro	Poco Provable	Posible	Muy Provable	Seguro
Consecuencias	Despreciable	Bajo	Bajo	Bajo	Medio	Medio
	Menor	Bajo	Bajo	Medio	Medio	Medio
	Moderado	Medio	Medio	Medio	Alto	Alto
	Mayor	Medio	Medio	Alto	Alto	Muy Alto
	Catastrofico	Medio	Alto	Alto	Muy Alto	Muy Alto

Figura 3.1: Matriz Consecuencias - Probabilidad

Identificador	R-01
Riesgo	Periodización inadecuada de tareas
Descripción	Dar prioridad alta a tareas que no lo son llevando a gastar mas tiempo del necesario en tareas de baja prioridad
Probabilidad	Media
Plan	Reajustar tiempos de tareas menos importantes, utilizar el colchón de tiempo.
Impacto	Medio

Cuadro 3.5: Riesgo 01

Identificador	R-02
Riesgo	Estimaciones erróneas
Descripción	Calculo erróneo del tiempo de desarrollo de una tarea. Lo que puede llevar a retrasar el resto de las tareas
Probabilidad	Alta
Plan	Priorizar tareas importantes y reajustar tiempos. Utilizar colchón de tiempo
Impacto	Muy Alto

Cuadro 3.6: Riesgo 02

Identificador	R-03
Riesgo	Rediseño de alguno de los componentes o Cambios en Requisitos
Descripción	Cambio de los requisitos o funcionalidad de algún componente que lleve al rediseño de este
Probabilidad	Medio
Plan	Priorizar tareas, reajustar tiempos y volver a estructurar las tareas en el calendario
Impacto	Alto

Cuadro 3.7: Riesgo 03

Identificador	R-04
Riesgo	Posibles tareas no planificadas.
Descripción	Aparición de tareas que no estaban previstas en la planificación del proyecto
Probabilidad	Media
Plan	Reajustar dependencias de tareas y estructurar de nuevo el tiempo que se va a dedicar a cada tarea
Impacto	Alto

Cuadro 3.8: Riesgo 04

Identificador	R-05
Riesgo	Desarrollo de funciones software erróneas.
Descripción	Desarrollo de funcionalidades que no van a ser incorporadas o que no se ajustan al resultado deseado
Probabilidad	Baja
Plan	Priorizar tareas importantes y reajustar tiempos. Utilizar colchón de tiempo
Impacto	Medio

Cuadro 3.9: Riesgo 05

Identificador	R-06
Riesgo	Retardo en tareas debido a sucesos ajenos al proyecto.
Descripción	Aparición de problemas de salud o familiares del desarrollador, que lleven a posponer alguna de las fases del proyecto
Probabilidad	Media
Plan	Priorizar tareas importantes y reajustar tiempos. Utilizar colchón de tiempo
Impacto	Alto

Cuadro 3.10: Riesgo 06

Identificador	R-07
Riesgo	Errores producidos en el servidor donde se alojan las herramientas.
Descripción	Problemas de conexión, acceso o perdida de información en los servidores que alojan el proyecto
Probabilidad	Baja
Plan	Posponer tareas, llegando a reescribir la información en caso de perdida. Reestructuración del proyecto y uso del colchón de tiempo.
Impacto	Medio

Cuadro 3.11: Riesgo 07

Identificador	R-08
Riesgo	Falta de conocimiento en las tecnologías
Descripción	Desconocimiento no planeado de cierta tecnología, que necesite un periodo de aprendizaje
Probabilidad	Baja
Plan	Añadir tareas de aprendizaje de dicha tecnología y reestructurar el calendario del proyecto incluyéndolas
Impacto	Medio

Cuadro 3.12: Riesgo 08

Identificador	R-09
Riesgo	Problemas con el software de terceros
Descripción	Aparición de dependencias o errores no esperados en el software de terceros utilizado
Probabilidad	Baja
Plan	Solución de errores y reestructuración de tareas
Impacto	Medio

Cuadro 3.13: Riesgo 09

3.3 Plan temporal

Para la realización completa del proyecto se han estimado 9 iteraciones que coinciden con los Sprint, la duración es variable a lo largo del proyecto siendo las primeras de dos semanas y las ultimas de una semana. Cada sprint consta de una reunión, aunque las iteraciones empiezan los lunes las reuniones programas se efectuaran los martes debido a la disponibilidad de los participantes.

En la Tabla 3.14 se indica la fecha de inicio de cada uno de los Sprint y la duración determinada.

Sprint 1	Fecha inicio	Duración
Sprint 1	12 de abril	2 Semana
Sprint 2	26 de abril	2 Semana
Sprint 3	10 de mayo	2 Semana
Sprint 4	24 de mayo	2 Semana
Sprint 5	7 de junio	2 Semana
Sprint 6	21 de junio	1 Semana
Sprint 7	28 de junio	1 Semana
Sprint 8	5 de julio	1 Semana
Sprint 9	12 de julio	1 Semana

Cuadro 3.14: Sprints previstos, con la fecha estimada inicialmente

En la Tabla 3.14 se pone en relación los sprint con el desarrollo de las fases del proyecto.

Semana	Fecha inicio	Sprint	Fase
Semana 01	12 de abril	Sprint 1	Fase 1
Semana 02	19 de abril		
Semana 03	26 de abril	Sprint 2	Fase 2
Semana 04	3 de mayo		
Semana 05	10 de mayo	Sprint 3	
Semana 06	17 de mayo		
Semana 07	24 de mayo	Sprint 4	Fase 3
Semana 08	31 de mayo		
Semana 09	7 de junio	Sprint 5	
Semana 10	14 de junio		
Semana 11	21 de junio	Sprint 6	
Semana 12	28 de junio	Sprint 7	
Semana 13	5 de julio	Sprint 8	Fase 4
Semana 14	12 de julio	Sprint 9	

Cuadro 3.15: Relación Sprint-Fase

Parte II

Marco Conceptual y Contexto

Capítulo 4

Marco Conceptual

En este capitulo se detalla el marco conceptual entorno a los ficheros ELF, el lenguaje de programación RUST y algunos conceptos relacionados con el desensamblado y la representación del flujo de programa mediante grafos, que van a ser utilizados en el desarrollo del proyecto.

4.1 ELF

ELF Executable and Linkable Format [31], también llamado Extensibe Linking Format, fue publicado por primera vez en la especificación de ABI del sistema operativo SVR4, mas tarde en 1999 fue escogida por 86open como formato de archivo binario estándar para sistemas Unix con arquitecturas de procesador x86

ES un formato extensible, flexible y multiplataforma, siendo compatible con distintos endians y tamaños de dirección. Estas características han permitido que se halla usado en multitud de sistemas operativos con distintos hardware.

Es un formato de fichero estándar para ficheros ejecutables, código objeto, librerías compartidas y Core dumps. El tipo de archivo ELF está determinado por el valor de e-type descrito en la Tabla 4.1 en la cabecera del fichero ELF, el struct que define el cabecero de los ficheros ELF esta descrito en la sección Apéndices B.1.2.

Hay tres tipos principales de archivos objeto.

- **Archivo reubicable**: contiene código y datos. Usado para vincular con otros archivos de objeto para crear un archivo ejecutable o de objeto compartido.
- **Archivo ejecutable**: contiene un programa para su ejecución.
- **Archivo objeto compartido**: contiene código y datos. Pude ser usado para:
  - Procesarlo con otros archivos de objetos compartidos y reubicables para crear otro archivo de objetos
  - Combinado con un archivo ejecutable y otros objetos compartidos para crear una imagen de proceso.

4.1.1 Estructura ELF

Los ficheros objetos [2], creados por el ensamblador(assambler) y el editor de enlaces, son representaciones binarias de los programas destinados a ejecutarse directamente en un procesador.

La estructura de los ficheros ELF esta descrita en la figura 4.1 y consta de:

- Cabecero ELF (ELF Header): definido en la sección apéndices B.1. Este situado al principio del fichero y describe la organización de los datos dentro del fichero.
- Secciones (Sections): contienen los datos del fichero objeto: instrucciones, tabla de símbolos, información de reubicación, etc.
- Tabla de cabeceros del programa (Program Header Table): dice al sistema como se crea la imagen del proceso. Los ficheros usados para construir imágenes de proceso deben contenerla.
- Tabla de cabeceros de sección(Section Header Table): contiene la información que describe las secciones del fichero. Todas las secciones tienen una entrada en la tabla. Cada entrada contiene información como el nombre, tamaño , ubicación, etc.

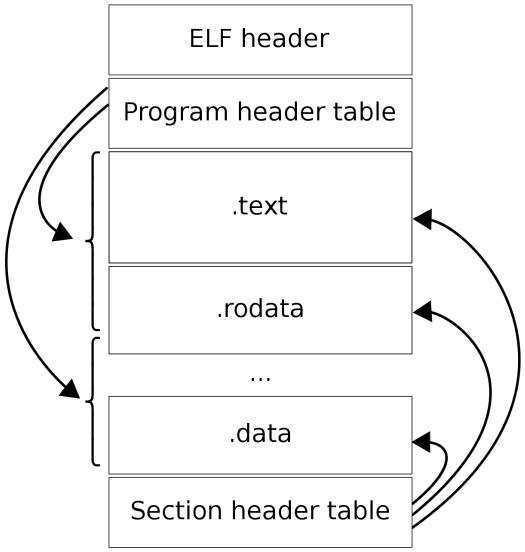

Figura 4.1: Estructura Archivo elf [23]

Nombre	Valor	Significado
ET_NONE	0x0	Sin tipo de archivo
ET_REL	0x1	Archivo reubicable (Relocatable)
ET_EXEC	0x2	Archivo ejecutable
ET_DYN	0x3	Archivo de objeto compartido (Shared object)
ET_CORE	0x4	Archivo core
ET_LOOS	0xfe00	Específico del sistema operativo
ET_HIOS	0xfeff	Específico del sistema operativo
ET_LOPROC	0xff00	Específico del procesador
ET_HIPROC	0xffff	Específico del procesador

Cuadro 4.1: Valores e-type elf

4.1.2 Secciones Comunes

En esta sección se describen las secciones comunes, que normalmente están incluidas en los ficheros ejecutables ELF [27]:

- **.bss(Block Started by Symbol)**: Esta sección contiene datos no inicializados que contribuyen a la imagen de la memoria del programa. Por definición, el sistema inicializa los datos con ceros cuando el programa comienza a ejecutarse.
- **.comment** Esta sección contiene información de control de versiones.
- **.data** y **.data1**: Estas secciones contiene datos inicializados que contribuyen a crear la imagen del programa en memoria.
- **.debug**: Esta sección contiene información para la depuración simbólica(Symbolic Debuggin).
- **.dynamic**:Esta sección contiene información sobre el enlazado dinámico(Dynamic Linkin)
- **.dynstr**: Esta sección contiene las cadenas necesarias para la vinculación dinámica, más comúnmente las cadenas que representan los nombres asociados con las entradas de la tabla de símbolos.
- **.dynsym**: Esta sección contiene la tabla de símbolos de enlace dinámico(Dynamic Linking Symbol Table).
- **.fini**: Esta sección contiene instrucciones ejecutables que contribuyen al código de terminación del proceso. Cuando un programa sale normalmente, el sistema se encarga de ejecutar el código de esta sección.
- **.got**: Esta sección contiene la tabla de compensación global.
- **.hash**: Esta sección contiene una tabla hash de símbolos.
- **.init**: Esta sección contiene instrucciones ejecutables que contribuyen al código de inicialización del proceso. Cuando un programa comienza a ejecutarse, el sistema se encarga de ejecutar el código de esta sección antes de llamar al punto de entrada del programa principal.
- **.interp**: Esta sección contiene el nombre de la ruta de un intérprete de programa.
- **.line**: Esta sección contiene información sobre el número de línea para la depuración simbólica, que describe la correspondencia entre la fuente del programa y el código de la máquina.
- **.note**: Esta sección contiene información en el formato de ”Sección de notas”.
- **.plt**: Esta sección contiene la tabla de vinculación de procedimientos.
- **.relNAME** y **.relaNAME**: Esta sección contiene información de reubicación como se describe a continuación.
- **.rodata** y **.rodata1**: Esta sección contiene datos de solo lectura que normalmente contribuyen a un segmento que no se puede escribir en la imagen del proceso.
- **.shstrtab**: Esta sección contiene los nombres de las secciones.
- **.strtab**: Esta sección contiene cadenas, más comúnmente cadenas que representan los nombres asociados con las entradas de la tabla de símbolos.
- **.symtab**: Esta sección contiene una tabla de símbolos.
- **.text**: Esta sección contiene las instrucciones ejecutables del programa.
- **.jcr**: Esta sección contiene información sobre clases de Java que deben estar registradas
- **.eh_frame**: Esta sección contiene información usado por C++

4.1.3 Secciones analizadas

De las secciones que suelen tener los archivos elf solo algunas contienen el código ejecutable. Las secciones que contienen el código son .init, .text y .finit, debido a que .init es el código de la secuencia de iniciación del proceso y .finit es el código que, en caso de ejecución correcta, cierra el proceso(secuencia de finalización) no son importantes a la hora de el flujo de ejecución del programa. Por lo que la principal sección a ser analizada es .data que contiene las instrucciones ejecutables del programa.

4.2 Lenguaje de programación: RUST

4.2.1 Lenguaje RUST

RUST es un lenguaje de programación multi-paradigma enfocado en el desempeño y en la seguridad, especialmente en la concurrencia segura. Proporciona uso seguro de memoria sin necesidad de utilizar un recolector de basura estas características están descritas en la Seccion 4.2.3.

4.2.2 Historia de RUST

RUST [21] comenzó como un proyecto paralelo de Graydon Hoare, un empleado de Mozilla. En poco tiempo, Mozilla vio el potencial del nuevo lenguaje y comenzó a patrocinarlo, antes de revelarlo al mundo en 2010.

A pesar de su relativa juventud, RUST ha aumentado constantemente en las filas de los lenguajes de programación populares. De hecho, aunque ocupó el puesto 33 en julio de 2019, en julio de 2020 había subido al puesto 18 en el índice de la comunidad de programación TIOBE. De manera similar, según la Encuesta para desarrolladores de Stack Overflow, RUST ha sido el lenguaje ”más querido”desde 2016.

4.2.3 Seguridad de memoria RUST

A menudo cuando se habla de programas se basa a seguridad de la memoria [11]. Esto significa que en cada ejecución que se realice no haya accesos invalidos a memoria. Dentro de la seguridad de memoria podemos encontrar distintos campos como: *use after free*, desreferencia de puntero nulo, uso de memoria no inicializada, *double free* o *buffer overflow*. Estas violaciones de memoria pueden ser usadas por atacantes para generar bloqueos inesperados de servicio o para alterar el comportamiento del programa.

Hay distintas formas de asegurar el uso seguro de la memoria, como los puntero inteligentes o los recolectores de basura. Estas medidas, aunque efectivas pueden afectar al rendimiento del sistema. Para obtener un uso de la memoria seguro, aparte de ayudar al desempeño, RUST utiliza la propiedad (*ownership*). El sistema de propiedad de RUST logra seguridad de memoria minimizando los costes de rendimiento.

Todo código escrito en RUST sigue ciertas normas de propiedad, que permiten al compilador administrarla sin incurrir en gastos mayores:

1. Cada valor tiene una variable, llamada propietario.
2. Solo puede haber un propietario a la vez.
3. Cuando el propietario se sale del alcance, el valor se eliminará. RUST libera la memoria asignada a esa variable
4. Los valores se pueden mover o tomar prestados entre variables. Estas reglas son aplicadas por una parte del compilador llamada verificador de préstamos.

Cuando se saca un valor de una variable, el propietario anterior deja de ser válido. Si el programador intenta utilizar la variable no válida, el compilador rechazará el código. Esto se puede evitar creando una copia profunda de los datos o utilizando referencias.

RUST no permite el uso de variables no inicializadas y punteros colgantes, lo que puede hacer que un programa haga referencia a datos no deseados. El compilador rastrea el alcance de las variables para garantizar que los "préstamos" sean válidos.

Respecto a los buffer overflow, una de las mitigaciones mas simples es requerir una verificación de límites al acceder a los elementos, lo cual agrega una penalización de rendimiento en tiempo de ejecución. Para solucionar esto en RUST los buffer integrados en la biblioteca estándar requieren de verificación de límites para cualquier acceso aleatorio, también proporcionan API de iterador que pueden reducir el impacto de estas verificaciones de límites en múltiples accesos secuenciales. Estos garantiza que lecturas y escrituras fuera de los límites sean imposibles para estos buffers.

Aprender sobre la seguridad de memoria y como el procesador "piensa" es muy útil a la hora de desarrollar código tanto seguro como eficiente.

4.2.4 Selección de Lenguaje RUST

Uno de los principales objetivos del equipo de desarrollo de RUST es garantizar el uso seguro de la memoria sin utilizar un recolector de basura [14]. Debido a que el uso inseguro de la memoria ha sido durante años un vector de entrada para atacantes en muchos lenguajes de programación. Esta gestión de memoria junto a al análisis estático del código incorporado en la base del lenguaje y el elaborado sistema de emisión de mensajes de error proporcionando consejos aplicables para solucionar los errores, hacen que los ejecutables y librerías escritas en RUST sea código de calidad, robusto y seguro.

El análisis estático es parte del lenguaje mismo; lo que es un dolor de cabeza es convencerlo de aceptar código dudoso. Y así es como RUST mejora la calidad de tu software: por la fuerza, ja También es considerado una combinación de lenguajes de alto nivel y de bajo nivel, siendo así un lenguaje adecuado para la de sistemas, proporcionando un desempeño similar o superior a C en programas similares.

RUST dispone de una gran biblioteca de librerías publicas que contiene código hecho y publicado por la comunidad en la que se encuentra numerables soluciones a problemas. Esta biblioteca se encuentra en crates.io, que en el momento de escribir este proyecto consta con mas de sesenta mil librerías publicadas.

Cargo es el administrador de paquetes de RUST. Es una herramienta que permite a los paquetes de RUST declarar sus diversas dependencias y garantizar que siempre obtendrá una compilación repetible.

Una de las principales ventajas de Cargo es que normaliza los comandos necesarios para la creación de programas y bibliotecas. El mismo comando puede ser usado para la compilación de distintos artefactos independientemente de sus nombres.

También evita la llamada directa a `rustc`, compilador de RUST que toma por entrada el código fuente del programa y produce como salida código binario ya sea como biblioteca o ejecutable, en su lugar se pueden hacer la llamada genérica a çargo build.[el] cual se ocupara de realizar la correcta llamada a `rustc` además de obtener automáticamente de un registro todas las decencias definidas anteriormente y de los arreglos necesarios para la compilación.

Para lograr los objetivos nombrados anteriormente Cargo hace cuatro cosas:

1. Introduce dos archivos de metadatos con varios bits de información del paquete.
2. Obtiene y crea las dependencias de su paquete.
3. Invoca `rustc` u otra herramienta de compilación con los parámetros correctos para compilar su paquete.
4. Introduce convenciones para facilitar el trabajo con paquetes de RUST.

Es solo una ligera exageración decir que una vez que sepa cómo construir un proyecto basado en Cargo, sabrá cómo construirlos todos.

4.3 Conceptos

4.3.1 Tipos de Código

- Código Fuente (Source code):

  El código fuente [35] es cualquier conjunto de código, escrito en lenguaje legible para humanos, este lenguaje esta diseñado para facilitar su desarrollo a los programadores, quienes mediante declaraciones especifican las accionéis que debe realizar la computadora.

  El código fuente a menudo es transformado, mediante compiladores o ensambladores, en código objeto para ser ejecutado o también puede ser interpretado y ejecutado sin necesidad de compilarlo, dependiendo del tipo de lenguaje.

- Código Objeto (Object code):

  Código objeto [34] hace referencia al código es el objeto o resultado de un proceso de compilación. Es una secuencia de declaraciones o instrucciones en lenguaje de computadora, generalmente código maquina o lenguaje intermedio. Es decir que provienen de la traducción de código fuente y se convierte en un fragmento del programa final y es específico de la plataforma de ejecución.

  Para que pueda ser utilizado el código objeto tiene que ser colocado en un archivo ejecutable, un archivo biblioteca o un archivo objeto. Los archivos objeto, a su vez, pueden ser vinculado para formar un archivo ejecutable o biblioteca. Un enlazador (linker) [47]es el programa encargado de juntar archivos de código objeto en un archivo ejecutable.

- Código intermedio, representación intermedia

  Estructura de dato o código que es utilizado internamente por un compilador o maquina virtual [32]como representación del código fuente, es utilizado por la mayoría de los compiladores modernos para la optimización, la traducción... Debe ser capaz de representar el código fuente sin perdida de información, independientemente de fuente o destino.

  El lenguaje intermedio [44] es un lenguaje de una máquina abstracta diseñada para ayudar a realizar el análisis de un programa, es usado por la mayoría de los compiladores modernos para la optimización, la

traducción...

El término proviene de la transformación en los compiladores donde transforman el código fuente en código intermedio para mejorarlo y posteriormente generar el código objeto o código maquina. Las diferencias fundamentales con el código maquinan son que cada instrucción representa exactamente una operación fundamental y la información de la estructura de control puede no estar incluida en el juego de instrucciones.

- Bytecode:

  Bytecode [29], es un conjunto de instrucciones diseñado para una ejecución eficiente por parte de un intérprete de software. Es tratado como un archivo binario que contiene un programa ejecutable similar al código objeto. Es utilizado para reducir las dependencias respecto del hardware específico y facilitar la interpretación. También es utilizado como código intermedio en un compilador. Los programas que usan este tipo de código suelen ser interpretados por una maquina virtual (interprete de bytcode).

  El nombre del código de bytes proviene de conjuntos de instrucciones que tienen códigos de operación de un byte, entre 0 y 255, seguidos de parámetros opcionales.

  Los traductores dinámicos traducen el bytecode a código máquina inmediatamente antes de su ejecución para mejorar la velocidad de ejecución.

  La mayor ventaja que ofrece el bytecode [37] es su portabilidad: el mismo bytecode puede ser ejecutado en diferentes plataformas y arquitecturas, siempre que consten de un interprete. El código de bytes a menudo se puede ejecutar directamente en una máquina virtual , o se puede compilar más en código de máquina para un mejor rendimiento.

- Lenguaje de máquina (Machine code):

  El lenguaje de máquina o código máquina [45] es el sistema de códigos directamente interpretable por la CPU. Lenguaje está compuesto por un conjunto de instrucciones que determinan acciones a ser tomadas por la máquina. Un programa consiste en una cadena de estas instrucciones más un conjunto de datos sobre el cual se trabaja. Estas instrucciones son normalmente ejecutadas en secuencia, pero el flujo del programa puede verse influenciado por instrucciones especiales de ”salto”que transfieren la ejecución a una dirección de memoria no secuencial. El código maquina es específico de la arquitectura de la máquina.

  Es un lenguaje de programación de bajo nivel [33], representación de nivel más bajo de un programa de computadora compilado o ensamblado, que se utiliza para controlar directamente la unidad central de procesamiento (CPU) de una computadora, es estrictamente numérico que está destinado a ejecutarse lo más rápido posible. Cada instrucción hace que la CPU realice una tarea muy específica.

  Lenguaje de programación primitivo y dependiente del hardware. Rara vez se escriben directamente en código de máquina en contextos modernos, debido a su complejidad y escasa legibilidad.

  El código fuente es traducido a código de máquina ejecutable por utilidades como compiladores, ensambladores y enlazadores, con la excepción de los programas interpretados que no se traducen a código de máquina.

- Código ensamblador (Assembly code):

  Lenguaje ensamblador (asm) [28], también denominado código máquina simbólico, es cualquier lenguaje de programación de bajo nivel en el que existe una correspondencia muy fuerte entre las instrucciones en el lenguaje y las instrucciones del código de máquina de la arquitectura, normalmente relación 1:1.Debido a esta dependencia cada lenguaje ensamblador está diseñado específicamente una arquitectura.

  El código de ensamblaje se convierte en código de máquina ejecutable mediante un programa de utilidad denominado ensamblador.

4.3.2 Desensamblado y Decompilado de código

Un desensamblador difiere de un decompilador, en que este tiene como objetivo un lenguaje de alto nivel en vez de al lenguaje ensamblador. la salida de un desensamblador, el desensamblado, es a menudo formateada para la legibilidad humana en vez de ser adecuada para la entrada a un ensamblador, haciendo que este sea principalmente una herramienta de ingeniería inversa. Existen algunos programas como GHIDRA o IDA que incluyen ambas funcionalidades.

Desensamblado, De código maquina a código ensamblador

Desensamblado [40] describe la operación de tomar un archivo ejecutable y producir como salida un código en ensamblador referente a dicho archivo. El programa que lleva a cabo el proceso se denomina desensamblador.

El código fuente en lenguaje ensamblador generalmente permite el uso de constantes y comentarios del programador. Estos son generalmente eliminados, por el ensamblador, del código ensamblado a código de máquina. De esta manera, durante el proceso de conversión de código ensamblador a código maquina conocido como ensamblado, se eliminan constantes y comentarios, debido a esto un desensamblador no puede rescatar los nombres de las variables o las funciones nombradas por el programador, comentarios ni código fuente. Algunos desensambladores hacen uso de la información de depuración simbólica presente en los archivos objeto tales como el ELF, añadiendo constantes e información adicional.

El desensamblado no es una ciencia exacta, es posible para un simple programa tener dos o más desensamblados razonables. Algunos de los desensambladores mas famosos son Interactive Disassembler (IDA), OllyDbg y radare2.

Decompilación, de código ensamblador a código fuente

Un decompilador [17] es un programa que recupera el código fuente de archivos ejecutables. Operación inversa a la compilación, Consiste en convertir el código máquina Código fuente.

No hay decompiladores perfectamente operacionales [39], el éxito de la decompilación depende de la cantidad de información presente en el código que está siendo decompilado y en la sofisticación del análisis realizado sobre él. Los formatos de bytecode utilizados por muchas máquinas virtuales en ocasiones incluyen metadatos en el alto nivel que hacen que la decompilación sea más flexible. Los lenguajes máquina normalmente tienen mucho menos metadatos, y son por lo tanto mucho más difíciles de compilar.

4.3.3 Bloques Básicos

En el entorno de los compiladores, un bloque básico [36] es un conjunto de instrucciones lineales y sin ramificaciones. Los compiladores normalmente descomponen los programas en bloques básicos como primer paso en el proceso de análisis. Los bloques básicos forman los nodos en un gráfico de flujo de un programa.
El código en un bloque básico tiene un punto de entrada, lo que significa que no hay código dentro de él es el destino de una instrucción de salto en cualquier parte del programa. Y un punto de salida, lo que significa que solo la última instrucción puede hacer que el programa comience a ejecutar código en un bloque básico diferente.

Un bloque básico [9] es una secuencia de código en línea recta con un solo punto de entrada y solo una salida. En GCC, los bloques básicos se representan mediante el tipo de datos basic_block.

El bloque básico [22] es una secuencia de código de línea recta que no tiene ramificaciones de entrada y salida excepto a la entrada y al final, respectivamente. El bloque básico es un conjunto de declaraciones que siempre se ejecutan una tras otra, en una secuencia.

4.3.4 Formato de Datos de Depuración

El Formato de Datos de Depuración (Debugging data format) es una forma de almacenar información sobre un programa compilado para ser usada por los depuradores de alto nivel. Almacena información de variables, tipos, constantes, subrutinas, etc. para que pueda ser traducida al desensamblar un programa.
La información es generada por el compilador y almacenada dentro del archivo ejecutable o biblioteca dinámica por el enlazador.
A continuación, describimos algunos de los formatos de depuración mas comúnmente usados:

- **DWARF [6]**:

  Formato desarrollado junto a ELF [38], pero independiente del mismo. Utiliza la estructura de datos DIE (Debugging Information Entry) formando una estructura en árbol. Un atributo DIE puede referirse a cualquier otro del árbol.

  Existen 2 tablas: Line Number Table, que asigna ubicaciones de código a ubicaciones de código fuente y viceversa, también especifica qué instrucciones son parte de prólogos de funciones y epílogos y la tabla Call Frame Information permite a los depuradores localizar frames en la pila de llamadas. Para ahorrar espacio, son representados como instrucciones bytecode para máquinas de estado finito simples

- **STAB**:

  El nombre proviene de las cadenas de la tabla de string [5], ya que los datos de depuración se guardaron originalmente como string en la tabla de símbolos del archivo de objeto a.out de Unix. Codifica las instrucciones en strings.

  Ha evolucionado de formato simple a formato un de depuración bastante complejo y más que consistente. No esta estandarizado ni bien documentadas. Sun Microsystems ha realizado una serie de extensiones para las puñaladas. GCC ha realizado otras extensiones, mientras intentaba aplicar ingeniería inversa a las extensiones de Sun.

- **COFF**:

El formato COFF [5] (Common Object File Format) es usado en archivos ejecutables, código objeto y bibliotecas compartidas, usada en sistemas Unix, introducido en Unix System V y ampliamente usado antes de antes de ser reemplazado en gran medida por ELF. Es la base para especificaciones extendidas como XCOFF y ECOFF. COFF y sus variantes siguen siendo usados en algunos sistemas Unix-like, en Microsoft Windows, en entornos EFI y en algunos sistemas embebidos.

- **XCOFF**:

El formato de archivo de objeto común extendido (XCOFF) [12] es definido por IBM y utilizado en AIX. XCOFF combina COFF con el concepto de formato de módulo TOC proporcionando vinculación dinámica y reemplazo de unidades dentro de un archivo de objeto.

Se utiliza una variación de XCOFF para archivos de objetos de 64 bits y archivos ejecutables. Es la definición formal de objeto de imagen de máquina y archivos ejecutables. El nombre predeterminado para un archivo ejecutable XCOFF es a.o.

- **OMF [18]**:

Llamado módulo de objeto reubicable (OMF:Relocatable Object Module Format) [48] se utiliza principalmente para software destinado a ejecutarse en microprocesadores Intel 80x86. Fue lanzada por Intel en 1981 bajo el nombre Object Module Format, y es el formato de archivo objeto más importante en DOS, Windows de 16 bits y OS / 2 de 16 y 32 bits.

Define información de nombre público y número de línea para depuradores y también puede contener datos de depuración en formato Microsoft CV, IBM PM o AIX. Pero solo proporciona soporte rudimentario para depuradores.

4.3.5 Tipos de gráficos de representación de un programa

- Gráficos de control de flujo (Control Flow Graph):
Representación esquemática , en forma de grafo dirigido, de todos los caminos , representados como nodos, que pueden ser atravesados a través de un programa durante su ejecución. Las relaciones entre nodos representadas mediante arcos son los distintos caminos de ejecución que puede seguir el programa.

- Gráficos de llamadas (Call Graph):
Un gráfico de llamadas representa las relaciones de llamadas entre subrutinas en un programa de computadora. Cada nodo representa corresponde a una función o procedimiento y cada relación la llamada de un procedimiento a otro. Los ciclos dentro del grafo representas llamadas recursivas.

- Gráfico de dependencias (dependence graph):
En matemáticas, informática y electrónica digital, un gráfico de dependencia es un grafo dirigido que representa las dependencias de varios objetos entre sí.

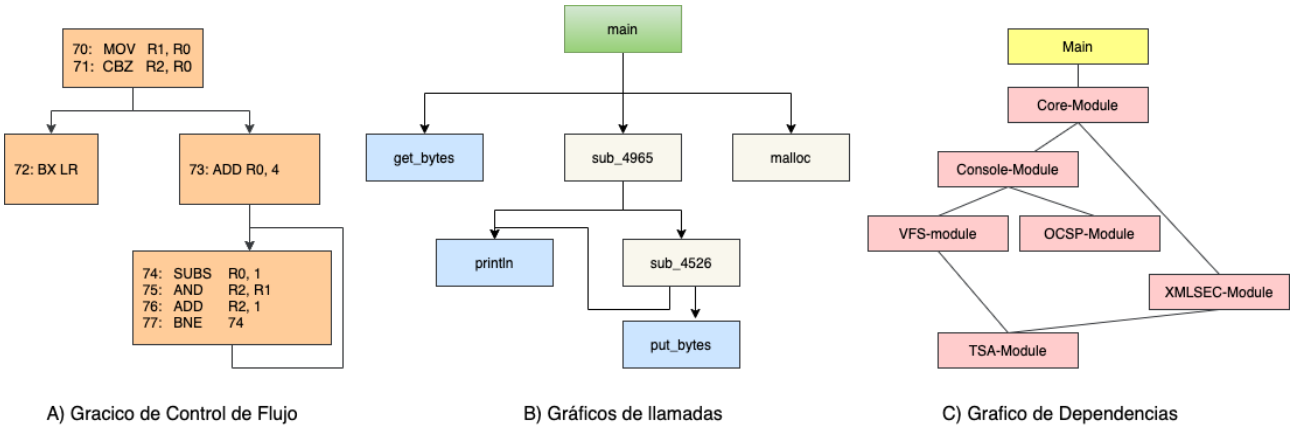

Figura 4.2: Tipos de Gráficos de Representación del Flujo de un Programa

4.3.6 Diagramas de Control de Flujo

Un diagrama de control de flujo [25] es un diagrama que describe un procesos, sistema o algoritmo informático mostrando todas las posibles vías de ejecución, pero sin mostrar cual será la vía que se ejecutara ni el orden en el que se encontraran.

Son usados en diversos campos [19] para documentar, estudiar, planificar, etc. Comúnmente son usados para representar procesos complejos mediante diagramas claros y fáciles de comprender.

Emplean diversas figuras geométricas (rectángulos, óvalos, diamantes ...) para representar el tipo de acción y flechas para unirlos y formar el flujo de ejecución.

4.3.7 Representación de Grafos de forma Gráfica (.dot)

DOT [30] es un lenguaje descriptivo en texto plano usado para representación de gráficos . Los ficheros de DOT suelen usar la extensión .gv o .dot. Aunque .dot esta en desuso debido a también es usada por Microsoft Word para plantillas.

Existen diversos programas para procesar DOT. Dependiendo si se quiere renderizan la descripción y generan gráficos (OmniGraffle, dot, neato, twopi, circo, fdp y sfdp) en diversos formatos de imagen, ejecutar cálculos sobre los grafos (gvpr, gc, accyclic, ccomps, sccmap y tred) o proveer una interfaz para editarlos (GVedit, KGraphEditor, lefty, dotty o grappa). Aunque mayoría forman parte del paquete Graphviz o lo usan parcialmente para ejecutar cálculos.

Graphviz [13] [42] es un programa de visualización de gráficos de código abierto. Describen descripciones de gráficos en un lenguaje de texto simple y crean diagramas en diversos formatos como SVG, PDF, Postscript, PNG, etc. o pueden mostrarlo en un navegador de gráficos interactivo. Incluye características útiles para diagramas concretos, como opciones de colores, fuentes, diseños de nodos tabulares, estilos de línea, hipervínculos y formas personalizadas.

Graphviz esta compuesto por un conjunto de herramientas y librerías que pueden generar o procesar archivos DOT. Entre las herramientas proporcionas se encuentran las siguientes:

- **dot**: Herramienta de línea de comandos para producir imágenes en capas de grafo dirigido. Esta es la herramienta predeterminada para usar si los bordes tienen direccionalidad.
- **neato**:Herramienta para usar si el gráfico no es grande y no sabe nada más al respecto.
- **fdp**: Motor de diseño para grafos.

- **sfdp**: Motor de diseño para grafos grandes. Versión multiescala de fdp para el diseño de gráficos grandes.
- **twopi**: Motor de diseño para esquemas de grafos radiales. Los nodos se colocan en círculos concéntricos dependiendo de su distancia al nodo raíz.
- **circo**: Motor de diseño para esquemas de gráficos circulares. Adecuado para diagramas de múltiples estructuras cíclicas, como algunas redes de telecomunicaciones.
- **dotty**: GUI para editar y visualizar gráficos DOT.
- **lefty**: Herramienta programable para mostrar DOT.

4.3.8 Grafos

En el ámbito de las matemáticas se describen como un [24] diagrama que representa mediante puntos y líneas las relaciones entre pares de elementos y que se usa para resolver problemas lógicos, topológicos y de cálculo combinatorio.

Representación simbólica de los elementos constituidos de un sistema o conjunto, mediante esquemas gráficos, conjunto no vació de elementos (vértices o nodos) unidos por un multi-conjunto de pares ordenados (aristas o arcos). permiten estudiar las interrelaciones entre unidades que interactúan unas con otras.

Grafo no dirigido

Los grafos no dirigidos [10] están formados por un conjunto de vértices y un conjunto de aristas que los unen de forma no direccional, es decir, que se pueden recorren indistintamente en ambos sentidos

Grafo Dirigido

Los grafos dirigidos o digrafos [26] son grafos orientados, constan de un conjunto de vértices y un conjunto de aristas que los une, pero en este caso las aristas son unidireccionales y son representadas con una flecha hacia el nodo destino. Estar aristas pueden ser entrantes o salientes para el nodo dependiendo de la dirección

4.3.9 Representación matemática de grafos

- Listas de aristas:

  Una lista de aristas [3] es una forma sencilla de representar un grafo la cual se compone solo de una lista en la que cada ítem corresponde a una arista. Para representar la arista se utiliza una tupla correspondiente a los nodos que une. Si es un grafo ponderado se añade un tercer valor a la tupla correspondiente al peso de la arista. Debido a que cada elemento del vector correspondiente a cada arista contiene solo dos o tres números el espacio total para almacenar la lista es de $\Theta(E)$ para $|E|$ aristas.

- Matriz de adyacencia :

  La matriz de adyacencia [3] es una matriz cuadrada que se utiliza como una forma de representar relaciones binarias.

  Todo grafo simple puede ser representado con este método. Para construir la matriz de adyacencia, cada elemento aij vale 1 cuando haya una arista que una los vértices i y j. En caso contrario el elemento aij

vale 0.La matriz de adyacencia, por tanto, estará formada por ceros y unos. Si es un grafo no dirigido la matriz de adyacencia es una matriz simétrica y si es un grafo ponderado se representa poniendo el peso de la arista en vez de un 1. Un matriz de adyacencia con | V | nodos ocupa un espacio $\Theta(V^2)$. Para un grafo disperso 0, es utilizado mucho espacio para representar solo un pequeño numero de aristas.

- Lista de adyacencia:

Las listas de adyacencia [3] [46] son usadas para la representación de todas las aristas o arcos de un grafo mediante una lista. Si es un grafo no dirigido, cada entrada es un conjunto de dos vértices conteniendo los dos extremos de la arista correspondiente. Si el grafo es dirigido, cada entrada es una tupla donde un elemento es nodo fuente y el otro el nodo destino del arco correspondiente. No tienen por que estar ordenadas. En un grafo dirigido el numero de entradas en la lista es igual al numero de arcos que contiene le grafo.

Capítulo 5

Soluciones Existentes

A continuación, se van a enumerar algunos de los frameworks y herramientas existentes mas conocido en el ámbito del desensamblado y de la ingeniería inversa, que son capaces de generar gráficos de control de la ejecución de programas:

- Radare2:
  Radare 2 [20] es un framework de ingeniería inversa y el análisis de binarios. Esta compuesto por un conjunto de pequeñas utilidades ejecutadas desde línea de comandos. radare2 es la principal herramienta del framework, permite abrir una serie de fuentes como si fueran archivos simples y sin formato. Implementa una interfaz de línea de comandos avanzada para moverse por un archivo, analizar datos, desensamblar, parchear binarios, comparar datos, buscar, reemplazar y visualizar.

  Radare1 [4] fue una herramienta forense para hacer búsquedas en discos duros, pero rápidamente fueron integrándose mas funcionalidades hasta reescribirlo por completo como framework para el análisis de archivos binarios e ingeniería inversa llamado ahora radare2. En la actualidad es considerado uno de los mayores proyectos libres de seguridad informática creado en España.

  Puede usarse en análisis forenses, en depuración de ejecutables, para realizar fuzzing o exploiting. También permite soportar una gran cantidad de formatos de ficheros ejecutables, desde los conocidos COFF y ELF hasta formatos de consolas de vídeo juegos, todos los procesadores populares desde las familias de procesadores de Intel y ARM hasta los SuperH, además de familias de procesadores poco conocidos.

- IDA:
  Interactive Disassembler (IDA) [43] es un framework de ingeniería inversa y análisis de binarios. Como parte de sus utilidades tiene desensamblador, decompilador y debugger para ejecutables Windows PE, Mac OS X, Mach-O y Linux ELF. Como desensamblador es capaz de crear mapas de ejecución, realizando análisis automático de código usando referencias cruzadas entre secciones del código, conocimiento de parámetros de llamadas a la API y otras informaciones.

  Realiza mucho análisis automático del código, usando referencias cruzadas entre las secciones del código, conocimiento de parámetros de las llamadas del API, y otra información.

- Ghidra:
  Ghidra [41] es un framework de ingeniería inversa gratuita y de código abierto desarrollada por la NSA.

Como IDA permite realizar desensamblado, decompilado y debuggear archivos binarios de múltiples formatos. Esta escrito en Java Swing y el decompilador en C++, además permite añadir plugins de 3 escritos en Java o Python.

- JEB Decompiler:
  JEB es una herramienta de desensamblado y decompilado de código nativo y aplicaciones Android. ES capaz de decompilar código de 32 y 64 bits. Las salidas del código fuente y el desensamblado son interactivas y pueden ser refactorizadas. Se pueden desarrollar scripts y plugins para completar las funcionalidades ofrecidas por JEB.

- Avrora:
  Avrora [1]es un framework de simulación y análisis para programas escritos para el microcontrolador AVR. Consta de multitud de funcionalidades como un simulador, una infraestructura para monitorización del comportamiento del programa, herramientas de creación de perfiles, enlaces de depuración con GDB. También tiene la funcionalidad de producir grafos de control de flujo de ejecución en formato .dot. Estas funcionalidades hacen que sea un framework completo para programas del microcontrolador AVR

- Jakstab: Jakstab [16] es una herramienta capaz de traducir código maquina a lenguaje ensamblador sobre la marcha mientras realiza análisis estático del flujo de datos. El análisis es usado para la reconstrucción del flujo de control principal del programa.

También existen soluciones menos conocidos como pueden ser JD Decompiler un decompilador de java o Binary Ninja que es un framework diseñado para la ingeniería inversa, y junto a esta multitud de herramientas y framework que abordan el tea desde diferentes necesidades y puntos de vista.

Parte III

Desarrollo del Sistema

Capítulo 6

Análisis

En esta sección se expresa el sistema a desarrollar mediante los conceptos y requisitos encontrados durante el análisis de datos y el estudio previo realizado. El proyecto será descrito mediante la definición de los requisitos, el modelo de dominio y enumerando los casos de uso.

6.1 Requisitos

En esta sección se describen el análisis y especificación de requisitos, tanto los requisitos del usuario como las especificaciones del sistema y entorno en el que se utilizara la aplicación. La identificación de estos requisitos será usada para definición de la arquitectura que será usada en el diseño del proyecto.

6.1.1 Requisitos Funcionales

Los requisitos funcionales son aquellos que describe las funciones que debe cumplir el sistema, en los que cada función esta descrita por la entrada, comportamiento y salida del sistema. Para este proyecto se han encontrado y descrito los siguientes requisitos funcionales:

- **RF - 01: Aplicación de línea de comandos**
  El sistema consistirá en una aplicación de línea de comandos. El software será llamado desde terminal con un comando y argumentos.

- **RF - 02: Entorno de compilación Linux**
  El sistema contendrá las dependencias necesarias para ser compilado en sistemas Linux

- **RF - 03: Compilación con cargo**
  El sistema estará configurado para que su compilación sea realizada con la herramienta cargo, propia del lenguaje RUST.

- **RF - 04: Ficheros binarios ELF 64**
  El sistema será capaz de procesar ficheros binarios ELF compilados para arquitectura 64-bit. Estos ficheros para procesar serán pasados como parámetro al llamar al ejecutable desde terminal.

- **RF - 05: Lenguaje DOT como salida**
  La generación del gráfico de control de flujo será en lenguaje DOT, que contendrá el gráfico generado.

- **RF - 06: Formato de salida**
  El sistema será capaz de devolver la salida como fichero o por la salida estándar, dependiendo de los parámetros de entrada.

- **RF - 07: Lenguaje de programación** RUST
  El sistema estará escrito por completo en el lenguaje de programación RUST.

- **RF - 08: Ejecución sin parámetros**
  El sistema al ser lanzado sin parámetros de entrada se mostrará un mensaje de ayuda al usuario con las opciones disponibles y el formato correcto de lanzar la ejecución del programa

- **RF - 09: Realizar desensamblado**
  El sistema será capaz de desensamblar ficheros ELF y devolverlo.

- **RF - 10: Generación de comando**
  Con la compilación del sistema será automáticamente insertado en la variable PATH para poder ser lanzado sin necesidad de acciones extra

- **RF - 11: Análisis estático**
  El desensamblado y la obtención del grafo serán realizados mediante la ejecución de análisis estáticos del fichero.

- **RF - 12: Identificación de la arquitectura**
  El sistema será capaz de identificar la arquitectura para la que esta compilado el ejecutable y comprobar si es una de las aceptadas.

6.1.2 Requisitos No Funcionales

Los requisitos no funcionales describen características de funcionamiento con relación a la calidad del programa, como puede ser la seguridad o fiabilidad del sistema. En relación con esto se describen los siguientes requisitos no funcionales:

- **RNF - 01: Mensaje de ayuda**
  Si la entrada contiene errores en los argumentos, el sistema no ejecutara nada y mostrara por la salida estándar un mensaje de ayuda para poder ejecutar el programa correctamente.

- **RNF - 02: Facilidad de uso**
  El sistema debe ser fácil de usar y proporcionar ayuda al usuario para que realice su correcta ejecución.

- **RNF - 03: Manual de instalación**
  Debe poseer una documentación de instalación.

- **RNF - 04: Funcionalidades documentadas**
  Debe poseer una documentación de funciones que es capaz de realizar y como pueden ser realizadas.

- **RNF - 05: Cancelación de ejecución ante errores**
  Si se detecta algún error durante la ejecución, esta se deberá detener y mostrar un mensaje que en la medida de lo posible ayude a identificar el error.

6.2 Casos de uso

En esta sección se describen los casos de uso del proyecto. Los casos de uso describen las actividades que pueden ser llevadas a cabo, así como los actores que intervienen.

En el escenario del proyecto existe un único actor. El actor es denominado *Usuario* y es el encargado de interactúa con la terminal y el que indica los parámetros necesarios para que el programa sea ejecutado.

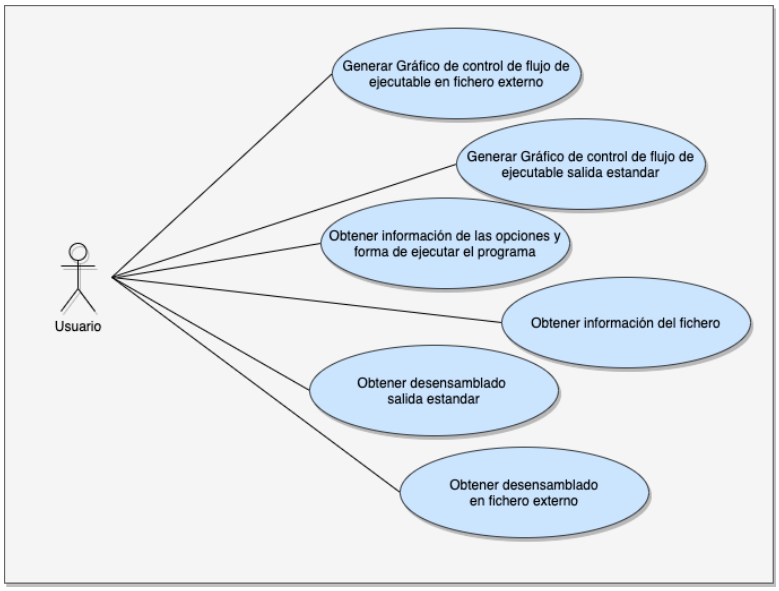

Figura 6.1: Diagrama de Casos de Uso

La Figura 6.1 describe la relación entre los actores y los casos de uso y estos han sido definidos en las Tablas 6.2, 6.1, 6.4 y 6.3.

Debido a que los casos de uso *Obtener desensamblado en fichero externo* y *Generar Gráfico de control de flujo de ejecutable en fichero externo* son muy similares a *Obtener desensamblado salida estándar* y *Generar Gráfico de control de flujo de ejecutable salida estándar*, no se ha realizado la tabla que los describe.

Caso de uso	Obtener opciones
Identificador	CU-02
Actores	Usuario
Pre-condiciones	Ninguna
Post-condiciones	Ninguna
Descripción	El usuario introduce el argumento destinado a generar la ayuda y el sistema imprime los posibles argumentos, su descripción y la forma de usarlos
Secuencia	1- El sistema es llamado por el usuario con el parámetro –help o -h
	2- El sistema comprueba los parámetros
	2.1- Si solo hay uno y es -h o –help continua el caso de uso
	3- El sistema imprime el mensaje de ayuda y el caso de uso finaliza.
Excepciones	2.a- El sistema al comprobar los parámetros detecta el parámetro -h y mas parámetros, el sistema termina la ejecución, imprime la forma de llamar al programa y el caso de uso termina

Cuadro 6.1: Caso de Uso CU-02

Caso de uso	Generar Gráfico de control de flujo
Identificador	CU-01
Actores	Usuario
Pre-condiciones	El fichero a procesar es un fichero ELF de 64-bits
Post-condiciones	El sistema a generado un gráfico de control de flujo del fichero de entrada.
Descripción	Generar el gráfico de control de flujo de un ejecutable ELF de arquitectura 64bits y devolverlo en formato .dot por la salida estándar.
Secuencia	1- El sistema es llamado por el usuario con la ruta de un fichero ELF como único parámetro
	2- El sistema comprueba el resto de los parámetros y carga la configuración indicada por los mismos
	3- El sistema Comprueba si el argumento es un fichero
	3.1- Si el argumento es un fichero lo abre
	4- El sistema realiza un análisis estático y genera los bloques básicos y la matriz de adyacencia asociada.
	5- El sistema toma la lista de bloques básicos y la matriz de adyacencia y genera el grafo asociado.
	6- El sistema toma el grafo y lo traduce a lenguaje dot
	7- Imprime el resultado por la salida estándar (terminal) y el caso de uso finaliza.
Excepciones	3.1.a- El sistema no es capaz de abrir el fichero, saca un mensaje de error con la descripción del error y el caso de uso finaliza

Cuadro 6.2: Caso de Uso CU-01

Caso de uso	Obtener información del fichero
Identificador	CU-04
Actores	Usuario
Pre-condiciones	Ninguna
Post-condiciones	Ninguna
Descripción	El usuario introduce el argumento destinado a obtener información del fichero
Secuencia	1- El sistema es llamado por el usuario con los parámetros para obtener información del fichero
	2- El sistema comprueba los parámetros
	2.1- Si son correctos continua el caso de uso
	3- El sistema imprime el mensaje de ayuda y el caso de uso finaliza.
Excepciones	2.a- El sistema imprime el error y la forma de llamar al programa y el caso de uso termina

Cuadro 6.3: Caso de Uso CU-04

Caso de uso	Generar desensamblado
Identificador	CU-03
Actores	Usuario
Pre-condiciones	El fichero a procesar es un fichero ELF de 64-bits
Post-condiciones	El sistema a generado el desensamblado del fichero.
Descripción	Generar el desensamblado de un ejecutable ELF de arquitectura 64bits y en instrucciones asm en formato DAWRF por la salida estándar.
Secuencia	1- El sistema es llamado por el usuario con la ruta de un fichero ELF como único parámetro
	2- El sistema comprueba el resto de los parámetros y carga la configuración indicada por los mismos
	3- El sistema comprueba si el argumento es un fichero
	3.1- Si el argumento es un fichero lo abre
	4- El sistema realiza un análisis estático y genera el código desensamblado de manera secuencial
	5- Imprime el resultado por la salida estándar (terminal) y el caso de uso finaliza.
Excepciones	3.1.a- El sistema no es capaz de abrir el fichero, saca un mensaje de error con la descripción del error y el caso de uso finaliza

Cuadro 6.4: Caso de Uso CU-03

6.3 Modelo de dominio

Representación conceptual de las clases candidatas que estén relacionadas con los requisitos descritos en el apartado 6.1. En el se muestran las clases y las relaciones existentes entre las mismas.

El modelo de dominio se muestra en la Figura 6.2 y sus clases descritas a continuación:

- **Path:**
  Representa la ruta a un fichero objetivo.
- **File:**
  Describe el fichero ELF, esta compuesto por las secciones y el cabecero de programa.
- **Section:** Forma parte de File y representa las secciones dentro del fichero ELF, consta del cabecero de sección y el código en bruto de la sección.
- **ProgramHeader:**
  Forma parte de File y describe el cabecero de programa y sus partes.
- **AdyacenceMatrix:**
  Estructura usada para representar las relaciones entre los nodos del grafo a generar.
- **BasicBlock:**
  Estructura que representa un conjunto de instrucciones que se ejecutan de forma lineal sin saltos.
- **Graph:**
  Estructura usada para la representación del grafo a obtener dentro del proyecto.

- **Decoder:**
  Elemento encargado de realizar la decompilación.

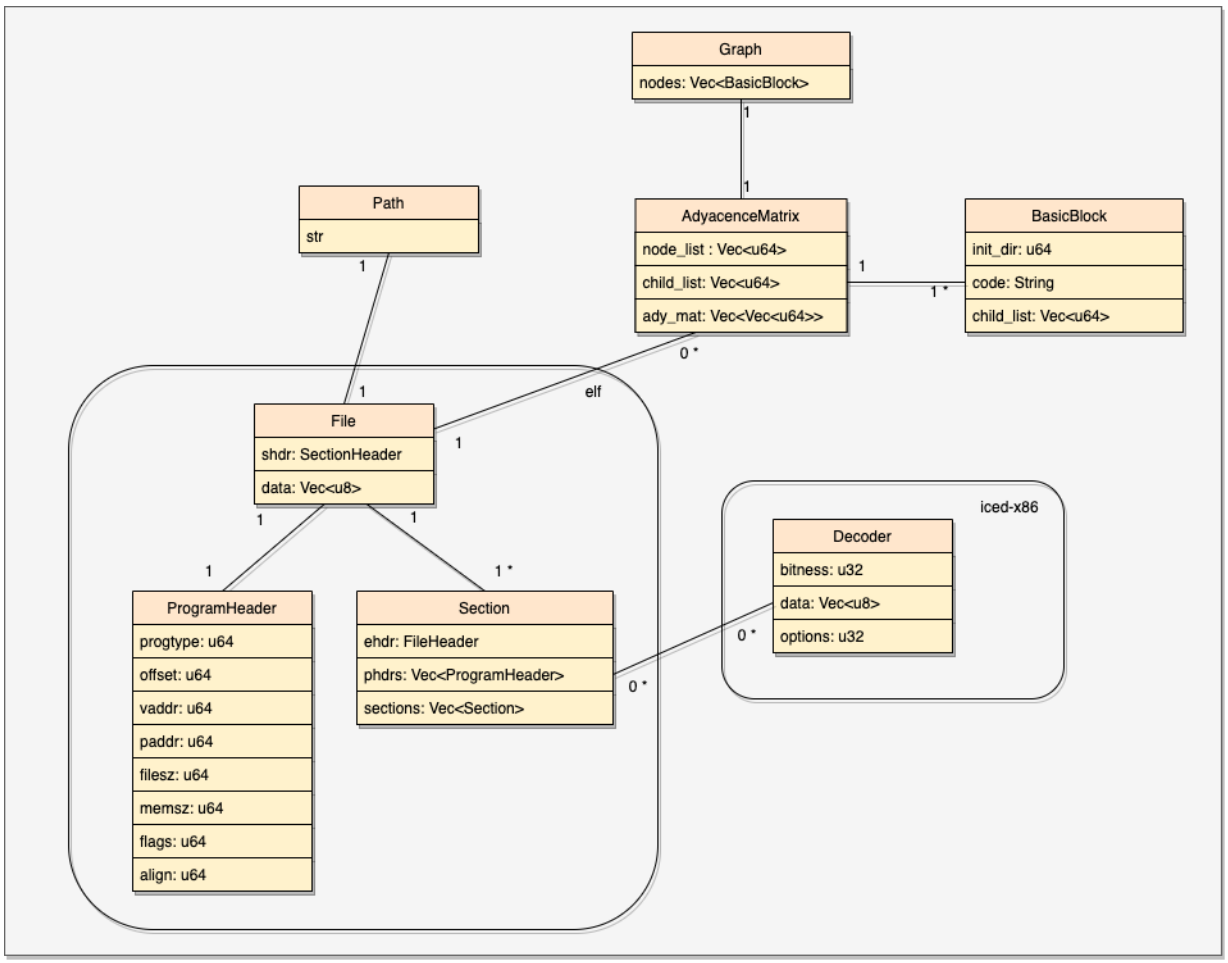

Figura 6.2: Modelo de Dominio

Capítulo 7

Diseño

7.1 Diseño

En este capitulo se presenta el diseño planteado para el proyecto. Que consiste en establecer la arquitectura, las interfaces, los algoritmos y las estructuras de datos necesarias para el proyecto. Esto se consigue a través de la traducción de los requisitos.

Para la descripción del diseño del proyecto se enuncia la arquitectura, que consiste en definición de alto nivel de la estructura del software, y los diagramas de secuencia y de flujo, que describirán el sistema y los algoritmos utilizados en la aplicación.

7.2 Arquitectura de la aplicación

Para el desarrollo arquitectónico de la aplicación se utilizará el patrón de diseño MVC. Debido a que es una aplicación de línea de comandos la vista estará representada por la terminal utilizada. La función main desempeñará la función de controlador y el modelo estará compuesto por las estructura y funciones utilizadas. La Figura 7.1 muestra una representación gráfica del patrón MVC con las conexiones usadas.

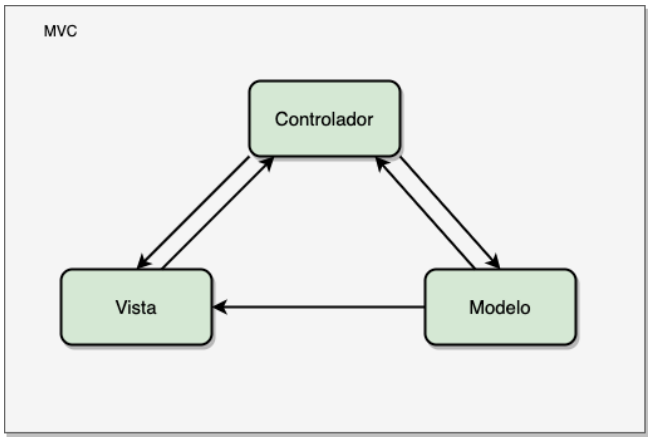

Figura 7.1: Diagrama Flujo Comprobación Tipo de Instrucción de Bifurcación

Siguiendo el patrón controlador, la función main ejercerá como intermediario entre la terminal(vista) y las funciones(modelo). El controlador recibirá los parámetros de entrada y dependiendo de estos realizará las llamadas necesarias a las funciones para posteriormente, cuando las llamadas terminen, devolver el resultado a usuario por el método descrito en los parámetros de entrada.

7.3 Diagrama de Flujo

En esta sección se incluyen los diagramas de flujo que representaran al sistema y algoritmos que describen el comportamiento del programa. Para este fin nos centramos en el diagrama de flujo general de la aplicación, que describirá el sistema, y el diagrama de flujo de la identificación de instrucciones de bifurcación, que es el algoritmo utilizado para identificar los bloques básico de código.

Para la realización de estos diagramas se ha seguido el código de formas geométricas y colores indicado en la leyenda, que esta descrita en la Figura 7.2.

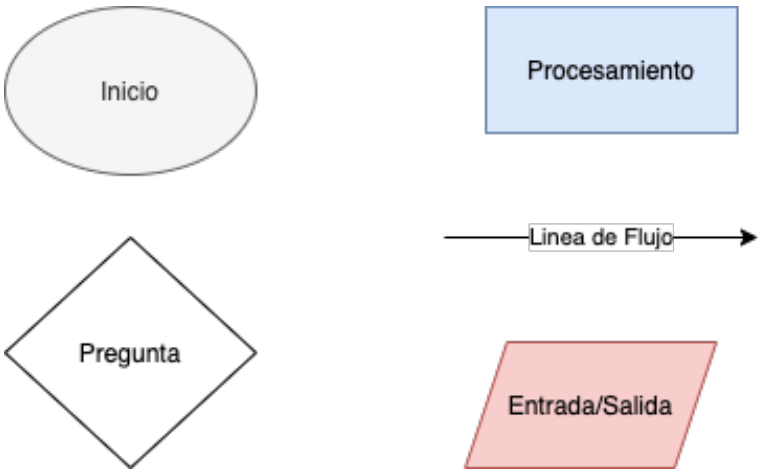

Figura 7.2: Leyenda Diagramas de Flujo

7.3.1 Diagrama de flujo general

En la Figura 7.3 se representa el diagrama de flujo de la aplicación, en el se describe el proceso de entrada, comprobación y carga de los parámetros de entrada, se indica el momento de apertura y procesamiento del fichero, y la comprobación de la forma de devolución de datos expresada en los parámetros de entrada.

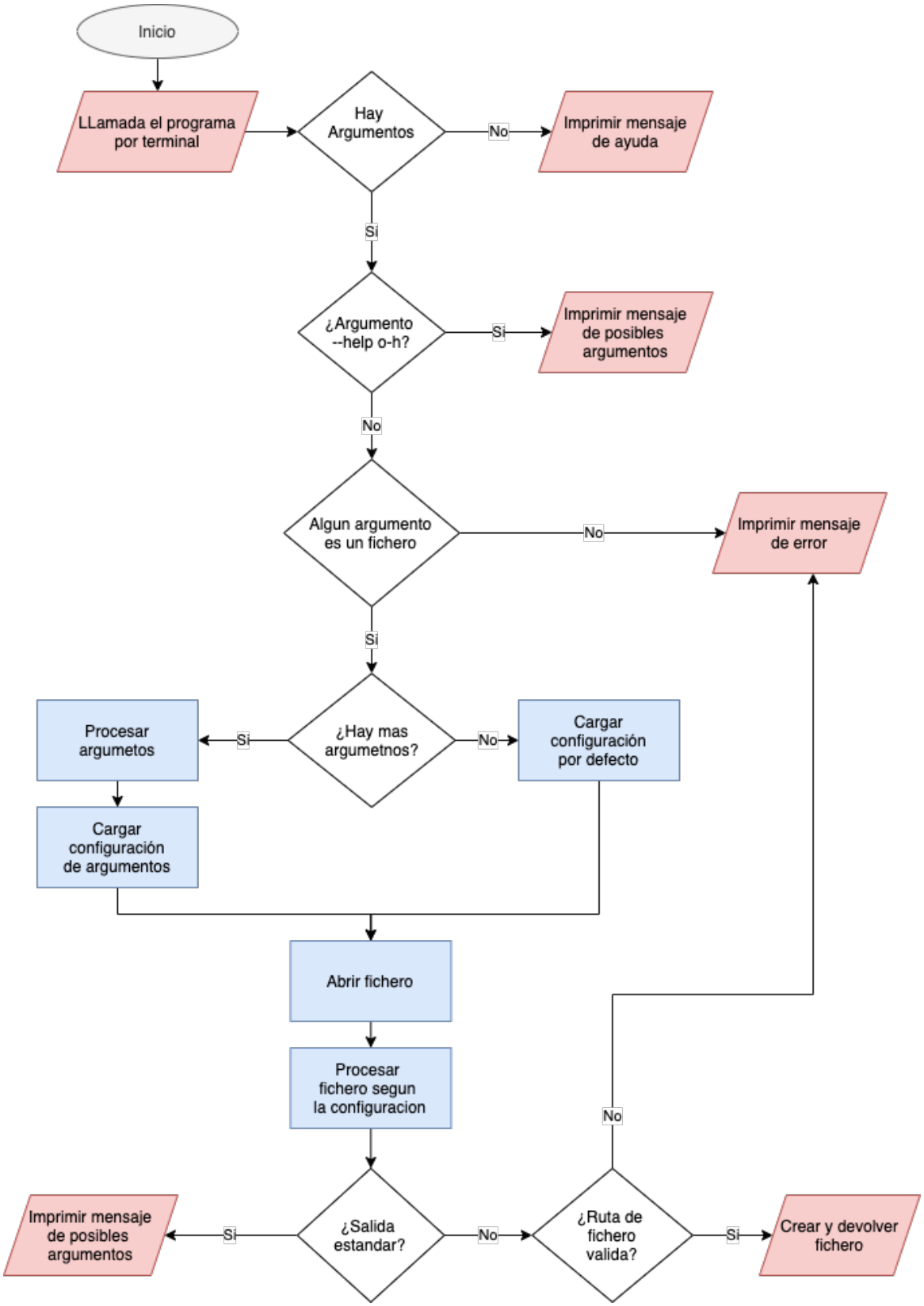

Figura 7.3: Diagrama Flujo general

7.3.2 Diagrama de flujo Comprobación tipo de instrucción

El diagrama de la Figura 7.4 representa el algoritmo utilizado la identifica ion de instrucción de bifurcación que será utilizado para la obtención de los bloques básicos. Este algoritmo utiliza los opcodes de las instrucciones para identificar el tipo de instrucción, y si coincide con alguna de las instrucciones de interés almacenar la dirección siguiente a la dirección de la instrucción para representar el fin de un bloque básico y el comienzo del siguiente.

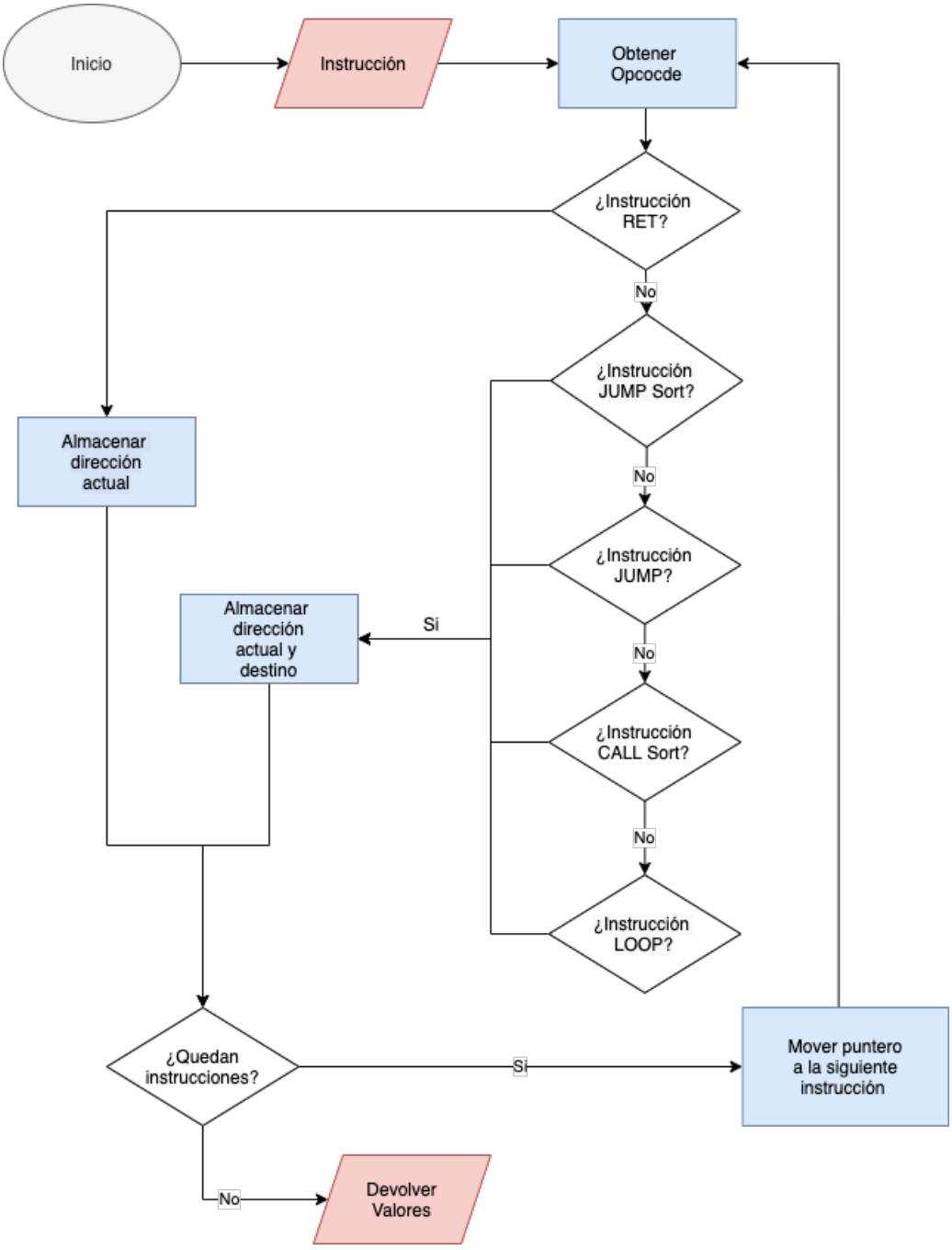

Figura 7.4: Diagrama Flujo Comprobación Tipo de Instrucción de Bifurcación

7.4 Diagrama de Secuencia

Los diagramas de secuencia definen las secuencias de interacciones de objetos entre si. En este apartad se muestra en la Figura 7.5 el diagrama de secuencia del caso de uso principal del proyecto, la generación de un grafo de control de flujo.

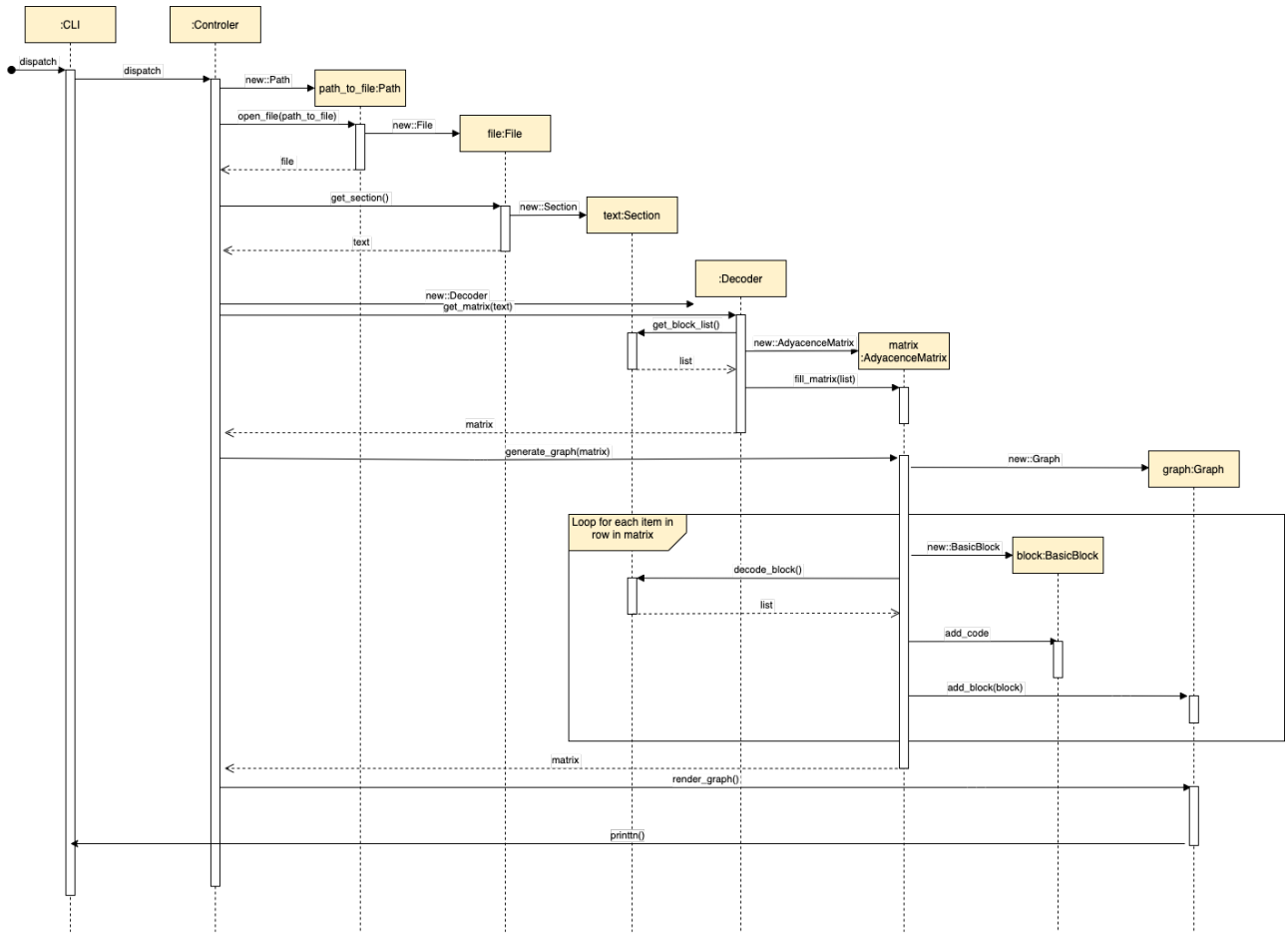

Figura 7.5: Diagrama de Secuencia del Caso de Uso General Grafo

Capítulo 8

Implementación

8.1 Entorno de desarrollo

El entorno en el que el software del proyecto va a ser desarrollado es Ubuntu montado en una maquina virtual proporcionada por la universidad. Mediante el comando ministrado a continuación obtenemos las especificaciones del sistema proporcionado.

```
1 uname -a
2 Linux virtual 5.4.0-72-generic #80-Ubuntu SMP Mon Apr 12 17:35:00 UTC 2021 x84_64
3 GNU/Linux
```

Para poder trabajar de forma cómoda desde se añadió en el servidor la clave publica de el ordenador desde el que se ha desarrollado

8.2 Herramientas de Desarrollo

La instalación y configuración de las herramientas y lenguajes se a descrito en la sección de ”Manual de instalación.[en] la sección de apéndices

- Lenguaje de programación usado: RUST
- Como IDE se ha utilizado Visual Code, en el cual se ha instalado los siguientes plugins
  - Remote SSH: para poder trabajar desarrollando de forma remota en el servidor.
  - RUST: Instalado en el servidor. Complemento para desarrollar RUST en Visual Code
  - RUST Syntax: Instalado en local. Analizador de código para corregir errores.
- Sistema operativo desde el que se trabaja en local es macOS BigSur
- La memoria ha sido escrita en el lenguaje LaTeXmediante el editor y compilador online Overleaf.

8.2.1 Herramientas web

También se ha utilizado una serie de portales web que proporcionan herramientas de rápido acceso, pero de gran utilidad a la hora de desarrollar el proyecto.

- play.rust-lang.org : Entorno donde probar código de RUST. Compila el código y lo ejecuta en el portal web, ha sido usado para realizar alguna prueba rápida de funcionalidad.
- dreampuf.github.io/GraphvizOnline: Compilador de lenguaje DOT online, permite usar diferentes motores de compilación y guardar los grafos en diversos formatos. Se ha usado en su mayor elección de la configuración visual de los grafos.
- programming-idioms.org: Proporciona formas estándar de usar distintos lenguajes de programación. Ha sido usado buscar algunas declaraciones en RUST y así agilizar el proceso de desarrollo.

8.2.2 Librerías externas

Para el desarrollo del proyecto se ha utilizado paquetes de terceros disponibles públicamente en el repositorio de paquetes de RUST crates.io. Se han seleccionado 2 paquetes debido a su utilidad.

El primero *elf* la versión 0.0.10, ultima disponible cuando se realizo el proyecto. Este paquete consiste en una librería completa para analizar archivos ELF. Este paquete ha sido usado para la descomposición en secciones de el código objeto.

El segundo paquete utilizado ha sido *iced-x86* versión 1.11.3. Biblioteca con las herramientas necesarias para poder escribir desensambladores, ensambladores y decodificadores, esta completamente escrita en RUST y proporciona soporte para archivos x64 y x86. Este paquete ha sido utilizado para crear el separador en bloques básicos y el desensamblador.

8.3 Implementación

8.3.1 Instrucciones de bifurcación

Los programas en lenguaje ensamblador se componen de instrucciones que corresponde al flujo de ordenes ejecutables. Y aunque una maquina es capaz de funcionar con un muy limitado juego de caracteres, las CPU modernas poseen un gran numero de instrucciones que son capaces de ejecutar. Esta instrucciones pueden ser clasificadas en e tipos según su función:

- Transferencia de datos.
- Aritméticas
- Control de flujo de programa
- Entrada y salida
- Lógicas

De estas nos vamos a centrar en las instrucciones de control de flujo de programa, estas instrucciones son las que cambian la ejecución normal del programa, la ejecución normal es lineal por lo que cuando una instrucción es ejecutada la siguiente en ejecutarse es la físicamente siguiente. A continuación, se describen las instrucciones de control de flujo tratadas en el proyecto.

Instrucciones solo con Opcode primario

- Instrucción RET:
  Opcode 0x3C, la instrucción ret transfiere el control a la dirección de retorno ubicada en la pila. Esta dirección generalmente se coloca en la pila mediante una instrucción de llamada.

- Instrucción Jump Sort:
  Opcode desde 0x70 al 0x7f,los saltos cortos son saltos cuyo objetivo está en el mismo módulo, intramodulares.
  Incluye las siguientes instrucciones: JO, JNO, JB, JNAE, JC, JNB, JAE, JNC, JZ, JE, JNZ, JNE, JBE, JNA, JNBE, JA, JS, JNS, JP, JPE, JNP, JPO, JL, JNGE, JNL, JG, JL, JN, JNL, JG

- Instrucción Jump:
  Opcode EA,E9, JMP,JMPF

- Instrucción Loop:
  Opcodes 0xE0, 0xE1, 0xE2, 0xE3. Instrucciones LOOP, LOOPE, LOOPZ, LOOPE, LOOPNE, LOOPNZ

- Instrucción CALL:
  Opcode 0xE8, Llamada a un procedimiento

Instrucciones con Opcode primario y mod R/W [7]

- Existe un grupo de instrucciones que tiene como Opcode primario FF (INC, DEC, CALLN, CALLF, JMPN, JMPF, PUSH), y no poseen Opcode secundario. Se selecciona mediante la observación de los bits 5 a 3 del byte Mod R/W. Cogiendo esos bit, siendo el bit de. la derecha el menos significativo, se pasa a decimal el valor y se usa como índice para el vector [INC, DEC, CALLN, CALLF, JMPN, JMPF, PUSH].

- Ejemplo:
  0xFFE0, pasamos 0xE0 a binario 1110 0000. Cogemos los bits del 3 al 5 100, 4 en decimal, corresponde a la instrucción JMPN.

Instrucciones con Opcode primario y secundario

- Instrucción Jump
  Opcode primario 0x0F, secundarios: desde 0x80 hasta 0x8F

- Instrucción JMPE
  Opcode primario 0x0F secundario 0xB8 , JMPE Jump to IA-64 Instruction Set
  Opcode primario 0x0F secundario 0x00 y byte Mod R/W 6 JMPE Jump to IA-64 Instruction Set

8.3.2 Estructura para representar grafos

En la elección de la representación del grafo vamos a comparar dos estructuras contrapuesta:

Por un lado, esta la matriz de adyacencia, que es eficiente a la hora de comprobar si existe una arista uniendo dos vértices, pero su principal desventaja es que requiere $\Omega(n^2)$ de memoria, aun cuando se tiene un grafo dirigido con menos de n^2 arcos. Además de el problema de memoria se añade que debido a que es una matriz es difícil añadir nuevos vértices y añadirlos requiere un alto coste computacional cuando se trabaja con matrices grandes.

Por el otro lado esta la lista de adyacencia, que elimina el problema del espacio de memoria, pero añade dificultad a la hora de comprobar la relación entre 2 nodos concretos. La lista de adyacencia de un vértice es una lista de todos los vértices adyacentes a dicho vértice.

8.3.3 Renderizar Grafos

Para la traducción de las estructuras creadas para representar grafos dentro del programa al lenguaje de dot se ha optado por la escritura de funciones propias en vez de utilizar librerías externas que pueden realizar dicho propósito como petgraph, debido a que tras probar su funcionamiento no encajan con la conversión que se buscaba, debido a que lo hacen de forma demasiado genérica para nuestra solución.

Se ha construido una función a la cual se le pasan las opciones de representación de nodos deseadas y la representación del grafo usada en el programa, un vector de bloques básicos y su lista de adyacencia, imprime los nodos y sus relaciones en el lenguaje dot.

Estructura del dígrafo en dot

La estructura de código dot que se ha seguido para la representación de grafos se muestra en la Figura: 8.1 mediante un código de ejemplo.

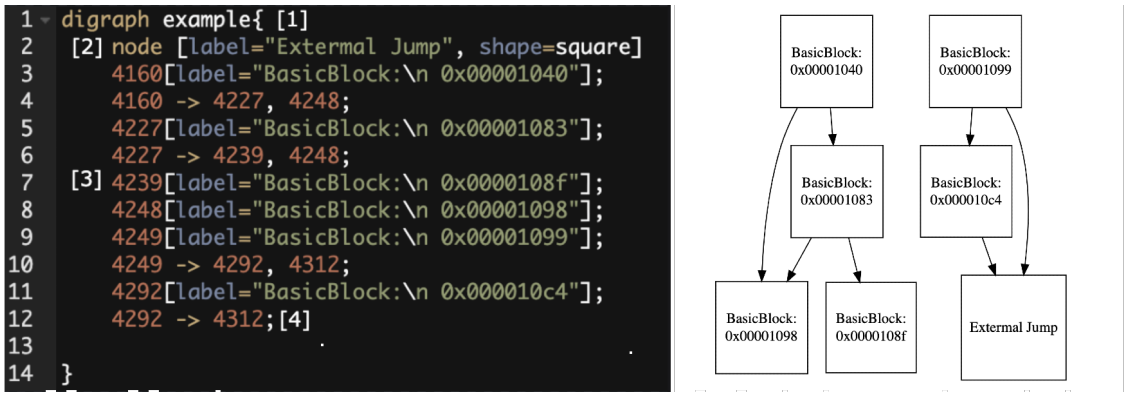

Figura 8.1: Código ejemplo de un grafo escrito en dot

- **[1]** Declaración del tipo de grafo, en nuestro caso como se representa el flujo mediante grafos dirigidos se utilizará la directiva ”digraph”seguida del nombre del grafo, en nuestro caso se utilizará el nombre del ejecutable a analizar.
- **[2]** Declaración de las propiedades generales de los nodos y aristas, para los nodos se usa la palabra *node* y *edge* para las aristas, seguidas de las variables y sus valores todo entre corchete.
- **[3]** Declaración de todos los nodos con su contenido.
- **[4]** Representación de las relaciones entre los nodos, a la izquierda el origen de la relación y a la derecha los nodos destino

8.3.4 Análisis estático de programas

El análisis del fichero se ha realizado mediante análisis estático. Este tipo de análisis es el realizado sin la ejecución del programa. En nuestro caso se ha realizado un análisis automático del código objeto.

En este proyecto el análisis estático consiste en la identificación de las secciones mediante los cabeceros de programa y de sección. Una vez identificadas las secciones y ubicadas dentro del fichero se extrae el código en bruto. El código en bruto consiste en una sucesión de bytes que representan las instrucciones, con estos bytes se identifican los opcode y con los opcode podemos identificar la instrucción y su tamaño. Una vez desensamblada la instrucción se mueve al byte siguiente a la instrucción y se vuelve a identificar el opcode. Esto se repite hasta terminar el código en bruto de la sección.

Capítulo 9

Pruebas

Debido a que se esta utilizando la metodología agile SCRUM las pruebas se han realizado durante el desarrollo de los distintos submódulos en los que se ha fragmentado el proyecto general. Para automatizar parte de las pruebas se ha escrito 2 fuzzer, que están descrito en la sección de Apéndices C.2, en el lenguaje de programación Python. Uno es un fuzzer simple que solo prueba el programa con distintas entradas validas, el segundo es un fuzzer algo mas complejo escrito para proporcionar entradas que pueden contener errores mediante arbitraria de bytes del fichero de entrada.

9.1 Casos de prueba

A parte de las pruebas realizadas con los fuzzers anteriormente nombrados se han realizado las siguientes pruebas:

9.1.1 Casos de prueba ejecución correcta

Identificador	CP-01
Descripción	Se pasará como argumento un fichero elf con arquitectura x64, como único argumento.
Entrada	Fichero elf x64
Salida Esperada	Grafo, escrito en lenguaje dot, devuelto por la salida estándar que represente el flujo del fichero de entrada

Cuadro 9.1: Caso de Prueba 01

Identificador	CP-02
Descripción	El programa será llamado sin ningún argumento
Entrada	Nada
Salida Esperada	Mensaje de ayuda para la llamada correcta del programa

Cuadro 9.2: Caso de Prueba 02

Identificador	CP-03
Descripción	El programa será llamado con un único argumento que será –help o -h
Entrada	–help o -h
Salida Esperada	Argumentos aceptados por el programa y la forma correcta de invocar al programa con los argumentos

Cuadro 9.3: Caso de Prueba 03

Identificador	CP-04
Descripción	Se invocará al programa con un fichero válido y el argumento –sections
Entrada	<Fichero elf x64>–sections o -S
Salida Esperada	Devuelve secciones de las que consta el programa programa

Cuadro 9.4: Caso de Prueba 04

Identificador	CP-05
Descripción	Probamos la funcionalidad de obtención de la cabecera de una sección.
Entrada	<Fichero elf x64>–secciones_header=<sección>o -s=<sección>
Salida Esperada	Cabecera de la sección seleccionada(Section Header)

Cuadro 9.5: Caso de Prueba 05

Identificador	CP-06
Descripción	Se probará si es capaz de generar un fichero con la salida esperada
Entrada	Fichero elf x64 y una ruta y nombre de fichero de salida validos. <Fichero elf x64><path del fichero de salida>
Salida Esperada	Gráfico del fichero elf escrito en dot en el fichero de salida indicado en la entrada

Cuadro 9.6: Caso de Prueba 06

Identificador	CP-07
Descripción	Se pasará como argumento un fichero elf con arquitectura x64.
Entrada	<Fichero elf x64>–dissasembly o -d
Salida Esperada	Desensamblado del fichero de entrada devuelto por la salida estándar

Cuadro 9.7: Caso de Prueba 07

Identificador	CP-08
Descripción	Se probará la ejecución con mas de un argumento valido
Entrada	-d <Fichero elf x64><path del fichero de salida>
Salida Esperada	Desensamblado del fichero de entrada en el fichero de salida

Cuadro 9.8: Caso de Prueba 08

9.1.2 Casos de prueba generan errores

Identificador	CP-09
Descripción	Se pasará como argumento un fichero elf con arquitectura x84.
Entrada	<Fichero elf x86>
Salida Esperada	Error, fichero no soportado y mensaje de ayuda

Cuadro 9.9: Caso de Prueba 09

Identificador	CP-10
Descripción	Se pasará como argumento un fichero ejecutable de Windows.
Entrada	<Fichero EXE>
Salida Esperada	Error, fichero no soportado y mensaje de ayuda

Cuadro 9.10: Caso de Prueba 10

Identificador	CP-11
Descripción	Fichero de otro formato, por ejemplo, un fichero de imagen
Entrada	<Fichero jpg>
Salida Esperada	Error, fichero no soportado y mensaje de ayuda

Cuadro 9.11: Caso de Prueba 11

Identificador	CP-12
Descripción	Se pasará como argumento una ruta de entrada no valida para el fichero elf a analizar
Entrada	<Fichero elf x64><path del fichero de salida>
Salida Esperada	Error, fichero no encontrado

Cuadro 9.12: Caso de Prueba 12

Identificador	CP-13
Descripción	Se pasará como argumento un fichero elf y una ruta inexistente de salida o sobre la que no se poseen permisos
Entrada	<Fichero elf x64><path del fichero de salida>
Salida Esperada	Error, ruta de salida no accesible

Cuadro 9.13: Caso de Prueba 13

Identificador	CP-14
Descripción	Se pasará como argumento un fichero elf y una ruta de salida hacia un fichero que ya existe
Entrada	<Fichero elf x64><path del fichero de salida>
Salida Esperada	Error, ruta de salida no accesible

Cuadro 9.14: Caso de Prueba 14

Capítulo 10

Conclusiones

En este capitulo se describen las conclusiones que han sido obtenidas durante el desarrollo del proyecto, los objetivos que han sido desarrollados y conseguidos y por ultimo el trabajo futuro y las siguientes versiones de la herramienta que serán llevadas a cabo.

10.1 Conclusiones

En este TFG se ha realizado una herramienta capaz de generar gráficos de control de flujo para ficheros objeto ELF compilados para arquitecturas de 64 bits mediante análisis estático y desensamblado. Se basa en la estructura y las secciones de los ficheros objeto para poder llevar a cabo su cometido. A continuación, se detallan las aportaciones conseguidas durante la realización de proyecto.

- Se ha analizado la estructura de los ficheros objeto que siguen el estándar ELF, la estructuras de datos usadas para almacenan la información y como están guardadas las instrucciones a ejecutar dentro del fichero. Gracias a esto se ha obtenido la información necesaria para la realización del proyecto.
- Se han analizado los distintos tipos de instrucciones, como identificarlas y las partes de las que están compuesta. Junto a esto se ha analizado el ïnstruction set"de AMD64 para poder identificar los opcodes de las instrucciones que se necesitan analizar y se ha descubierto el gran conjunto de instrucciones que los procesadores son capaces de procesar.
- Se han descubierto e investigado herramientas y frameworks destinados a la ingeniería inversa, descomposición, desensamblado y análisis de ficheros objeto de multitud de arquitecturas y estándares distintos. Que han proporcionado una idea general sobre las técnicas usadas para poder realizar la herramienta graphflow.
- Se ha realizado experimentación y aprendizaje sobre el lenguaje de programación RUST y sus herramientas, dejando claro su potencial como lenguaje de programación general.
- Se ha investigado sobre las diferentes formas de representación de grafos de manera matemática y su traducción a código.
- Se ha analizado el lenguaje de representación de grafos DOT, frameworks y programas capaces de procesar este lenguaje y los diferentes motores de representación.
- Se ha estudiado y aprendido la complejidad de las aplicaciones de terminal.

10.2 Trabajo futuro

Como no hay código sin vulnerabilidades parte del trabajo futuro será encontrarlas y arreglarlas. Como parte del trabajo futuro se enunciarán algunas de las posibles mejoras de la herramienta:

- Paralelización de las partes del programa que puedan aceptarla.
- Extender los tipos de archivos objetos aceptados, tanto de otras arquitecturas como de diferentes estándares.
- Transformación de aplicación de terminal a aplicación web usando la arquitectura API Rest.
- Construcción de interfaz gráfica de usuario, para convertir la herramienta en aplicación de escritorio. Integración con los motores proporcionados por Graphviz.

A parte de estas posibles mejoras la herramienta puede ser completada implementando características que ayuden al análisis de ficheros objeto en otros campos diferentes a la obtención de grafos, como puede ser el análisis de estructuras de datos o la obtención de código decompilado.

APENDICES

Appendices

Apéndice A

Manual de Usuario

A.1 Manual de instalación

Esta guía esta hecha para la instalación en ordenadores con Ubuntu Linux, con otros sistemas operativos la instalación de los componentes previos puede variar. A continuación, se describe la instalación de los componentes necesarios antes de la instalación:

Lenguaje de programación RUST

Para instalar RUST nos descargamos el script de instalación proporcionado por equipo de RUST y lo ejecutamos, mediante el comando:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Este script le mostrara 3 opciones como en la Figura A.1, para una instalación normal introducimos un 1 y continuamos

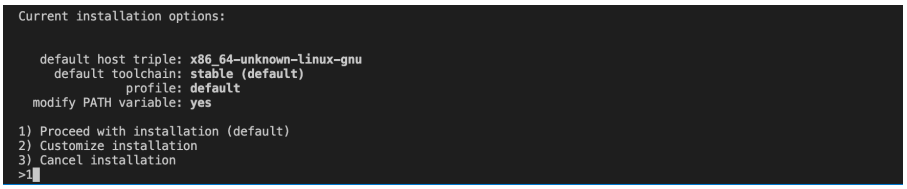

Figura A.1: Mensaje Opciones Instalación Script RUST

Este script instalara rls, rust-src, rust-analysis, cargo, clippy, rust-docs y rust-std.
Una vez el script ha terminado ya esta RUST instalado. Para poder usarlo hay que recargar la variable PATH, esto se puede hacer reiniciando la terminal o incluyendo el directorio de cargo en la variable PATH actual con el comando:

```
source $HOME/.cargo/env
```

A.1.1 Git para terminal

Para instalar Git en vamos a utilizar el gestor de paquetes de Ubuntu *apt*. Para realizar la instalación será necesario tener permisos de root o sudo.
Antes de realizar la instalación es recomendable asegurarse que todo este actualizado para evitar dependencias, una vez comprobado procedemos a la instalación de git.

```
1   sudo apt update
2   sudo apt install git-all
```

A.1.2 Instalación de la herramienta

El código de la aplicación esta alojado en gitlab.inf.uva.es, por lo que para acceder a el es necesario clonarlo desde el repositorio. Para hacer esto desde terminal se utilizará el siguiente comando:

```
1   git clone https://gitlab.inf.uva.es/marilop/tfgmariolopez.git
```

Para realizar la instalación del repositorio clonado, será necesario cambiar a la carpeta clonada y llamar a la herramienta cargo, que se encargará de resolver las dependencias del paquete, compilarlo y hacer que se pueda utilizar la herramienta.

```
1   cargo build
```

A.2 Pagina Man

```
NAME
       graphflow - genera grafico de control del fichero
SYNOPSIS
       graphflow       [--graph]
                      [--section_header=Section_Name]
                      [--all_section_header]
                      [--header]
                      [--disassembly]
                      [--type]
                      [--output=File]
                      [-h|--help]
                      objfile
DESCRIPTION
       graphflow imprime información o gráfico de control de
       flujo sobre el fichero objeto indicado.
       Las opciones indican la información a ser mostrada.

objfile   indica el fichero objeto a ser analizado.

OPTIONS
       Si no se proporciona ninguna de las opciones se to-
       mara por defecto la opción --graph

       --graph
              Imprime el grafo de control de flujo del fi-
              chero objeto indicado
       --section_header=Section_Name
              Devuelve el cabecero de la sección indicado
```

```
--all_section_header
       Imprime los cabeceros de todas las secciones
       contenidas en el fichero objeto
--disassembly
       Imprime el desensamblado del fichero objeto
--type
       Imprime el tipo de fichero objeto
--output=File
       Redirige la salida al fichero indicado
-h|--help
       Imprime esta sección de ayuda
```

Apéndice B

Manual del Desarrollador

```
pub struct AdyacenceMatrix{
    node_list: Vec<u64>,
    child_list: Vec<u64>,
    ady_mat: Vec<Vec<u64>>,
}
```

```
struct BasicBlock{
    init_dir: u64,
    code: String,
    child_list: Vec<u64>,
}
```

B.1 x64 instruction set

B.1.1 Codificación de instrucciones

Una instrucción se codifica como una cadena de entre uno y 15 bytes de longitud. La secuencia completa de bytes que representa una instrucción, incluida la operación básica, la ubicación de los operandos de origen y destino, cualquier modificador de operación y cualquier valor inmediato y/o de desplazamiento.

B.1.2 Struct de secciones

Para cualquier arquitectura de sistema el ejecutable debe incluir <sys/elf_common.h>y <sys/elf_generic.h>. Estos ficheros de encabezado describen la siguientes estructuras de datos e incluyen estructuras para secciones dinámicas, secciones reubicaras y tablas de símbolos.

Nombre	Tipo
Elf32_Addr	Unsigned 32-bit program address
Elf32_Half	Unsigned 16-bit field
Elf32_Lword	Unsigned 64-bit field
Elf32_Off	Unsigned 32-bit file offset
Elf32_Sword	Signed 32-bit field or integer
Elf32_Word	Unsigned 32-bit field or integer

Cuadro B.1: Tipos de datos utilizados en arquitecturas de 32-bit

Nombre	Tipo
Elf64_Addr	Unsigned 64-bit program address
Elf64_Half	Unsigned 16-bit field
Elf64_Lword	Unsigned 64-bit field
Elf64_Off	Unsigned 64-bit file offset
Elf64_Sword	Signed 32-bit field
Elf64_Sxword	Signed 64-bit field or integer
Elf64_Word	Unsigned 32-bit field
Elf64_Xword	Unsigned 64-bit field or integer

Cuadro B.2: Tipos de datos utilizados en arquitecturas de 64-bit.

Todas las estructuras de datos que el formato de fichero describe sigue el tamaño “natural” y las pautas de alineamiento para la clase relevante. Si es necesario las estructuras de datos contienen relleno(padding) explicito para asegurar el alineamiento de 4-bytes para objetos de 4-bytes o para forzar a los tamaños de estructura ser múltiplo de 4.

elf Header 64-bit

Listing B.1: elf Header 64 bits

```
1   #define EI_NIDENT 16
2 typedef struct {
3     unsigned char   e_ident[EI_NIDENT];
4     Elf64_Half    e_type;
5     Elf64_Half    e_machine;
6     Elf64_Word    e_version;
7     Elf64_Addr    e_entry;
8     Elf64_Off     e_phoff;
9     Elf64_Off     e_shoff;
10    Elf64_Word    e_flags;
11    Elf64_Half    e_ehsize;
12    Elf64_Half    e_phentsize;
13    Elf64_Half    e_phnum;
14    Elf64_Half    e_shentsize;
15    Elf64_Half    e_shnum;
16    Elf64_Half    e_shstrndx;
17 } Elf64_Ehdr;
```

Descripción de los valores de el encabezado del fichero ELF [8].

- **e_ident:** Indica que es archivo objeto.
- **e_type:** Indica el tipo de archivo de objeto

- **e_machine:** Especifica la arquitectura del archivo.
- **e_version:** Indica la versión del archivo de objeto.
- **e_entry:** Indica la dirección virtual de entrada.
- **e_phoff:** Desplazamiento del archivo de la tabla de encabezado del programa en bytes.
- **e_shoff:** Desplazamiento del archivo de la tabla de encabezado de sección en bytes.
- **e_flags:** Contiene indicadores específicos del procesador.
- **e_ehsize:** Tamaño del encabezado ELF en bytes.
- **e_phentsize:** Contiene el tamaño en bytes de una entrada en la tabla de encabezado del programa del archivo.
- **e_phnum:** Indica el número de entradas en la tabla de encabezado del programa.
- **e_shentsize:** Tamaño de encabezado de sección en bytes.
- **e_shnum:** Contiene número de entradas en la tabla de encabezado de sección.
- **e_shstrndx:** Contiene el índice de la tabla de encabezado de sección de la entrada asociada con la tabla de cadena de nombre de sección.

Section Header 64-bit

Listing B.2: elf Secction Header 64 bits

```
typedef struct {
    Elf64_Word    p_type;
    Elf64_Word    p_flags;
    Elf64_Off     p_offset;
    Elf64_Addr    p_vaddr;
    Elf64_Addr    p_paddr;
    Elf64_Xword     p_filesz;
    Elf64_Xword     p_memsz;
    Elf64_Xword     p_align;
} Elf64_Phdr;
```

Descripción de los valores de el encabezado sección de ficheros ELF [15].

- **sh_name:** Especifica el nombre de la sección.
- **sh_type:** Categoriza el contenido de la sección
- **sh_flags:** Compuesto por flags de 1 bit que indican características de la sección
- **sh_addr:** Dirección en la que el primer byte esta ubicado
- **sh_offset:** byte offset del principio del archivo al primer byte de la sección.
- **sh_size:** Este miembro tiene el tamaño de la sección en bytes.
- **sh_link:** Enlace de índice de tabla de encabezado de sección.
- **sh_info:** Información adicional, su interpretación depende de la sección.
- **sh_addralign:** Restricciones de alineación de direcciones, no lo tienen todas las secciones
- **sh_entsize:** Tamaño en bytes para cada entrada de la tabla de entradas de tamaño fijo si posee alguna.

B.1.3 Struct Utilizados

Listing B.3: Struct AdyacenceMatrix

```
pub struct AdyacenceMatrix{
    node_list: Vec<u64>,
    child_list: Vec<u64>,
    ady_mat: Vec<Vec<u64>>,
}
```

Listing B.4: Struct BasicBlock

```
struct BasicBlock{
    init_dir: u64,
    code: String,
    child_list: Vec<u64>,
}
```

Listing B.5: Struct Graph

```
struct Graph{
    node_list: Vec<BasicBlock>,
}
```

Apéndice C

Código externo a la aplicación

C.1 Código para pruebas

Se ha escrito un programa simple en el lenguaje de programación C y compilado con GCC para realizar pruebas durante el desarrollo. El programa se muestra en el Código C.1

Listing C.1: Código para Pruebas

```
1       #include <stdio.h>
2
3       int main(){
4               int a = 20;
5               for(int i=0; i<=0; i++){
6                       a=a-1;
7                       a*2;
8
9               }
10              return 0;
11      }
```

C.2 Código Fuzzers

En esta sección se presentan los fuzzer usados para hacer pruebas sobre el proyecto. Esta compuesto de 2 fuzzer, el primero Fuzzer C.2 toma los ficheros objeto dentro le la carpeta corpus dentro del proyecto, los ejecuta y muestra si se ha ejecutado correctamente. El segundo Fuzzer ?? es igual que le primero, pero cambia bytes aleatorios por otros generados aleatoriamente. Estos fuzzer están basados en los fuzzers desarrollados por Brandon Falk en ??.

Listing C.2: Código Fuzzer

```
1 import glob, subprocess, random, time, threading
2 def fuzz(thr_id: int, inp: bytearray):
3       assert isinstace(thr_id, int)
4       assert isinstace(inp, bytearray)
5
```

```
6       tmpfn = f"tmpinput{thr_id}"
7       with open(tmpfn, "wb") as fd:
8               fd.write(inp)
9
10      sp = subprocess.Popen(["./graphflow, tmpfn"],
11              stdout=subprocess.DEVNULL,
12              stderr=subprocess.DEVNULL)
13      ret = sp.wait()
14
15      if ret != 0:
16              print(f"Exited with {ret}")
17
18
19 corpus_filenames =glob.glob("corpus/*")
20 corpus = set()
21
22 for filenames in corpus_filenames:
23      corpus.add(open(filenames,"rb").read())
24
25 corpus =list(map(bytearray, corpus))
26 start = time.time()
27 cases = 0
28
29 def worker (thr_id):
30      global start, corpus, cases
31      while True:
32              fuzz(thr_id, random.choice(corpus))
33              cases += 1
34              elapsed = time.time() - start
35              fcps = float(cases) / elapsed
36              # fpcs = fuzzer cases per second
37              print(f"[{elapsed:10.4f}] cases {cases:10} | fcps {fcps:10.4f}")
38
39 for thr_id in range(192):
40      threadin.Thread(tarhet=worker, args=[thr_id]).start()
41
42 while threading.acite_count() > 0:
43      time.sleep(0.1)
```

Listing C.3: Código Fuzzer con mutación

```
1
2
3 import glob, subprocess, random, time, threading
4 def fuzz(thr_id: int, inp: bytearray):
5       assert isinstace(thr_id, int)
6       assert isinstace(inp, bytearray)
7
8       tmpfn = f"tmpinput{thr_id}"
9       with open(tmpfn, "wb") as fd:
10              fd.write(inp)
11
12      sp = subprocess.Popen(["./graphflow, tmpfn"],
13              stdout=subprocess.DEVNULL,
14              stderr=subprocess.DEVNULL)
15      ret = sp.wait()
16
17      if ret != 0:
18              print(f"Exited with {ret}")
19
```

```
20
21 corpus_filenames =glob.glob("corpus/*")
22 corpus = set()
23
24 for filenames in corpus_filenames:
25       corpus.add(open(filenames,"rb").read())
26
27 corpus =list(map(bytearray, corpus))
28 start = time.time()
29 cases = 0
30
31 def worker (thr_id):
32       global start, corpus, cases
33       while True:
34
35           # Inicio del mutacion del fichero de entrada
36               inp = bytearray(random.choice(corpus))
37
38               for _ in range (random.randint(1,8)):
39                       inp[random.randint(0,len(inp) - 1)] = random.randint(0,255)
40
41               fuzz(thr_id, inp)
42               cases += 1
43               elapsed = time.time() - start
44               fcps = float(cases) / elapsed
45               # fpcs = fuzzer cases per second
46               print(f"[{elapsed:10.4f}] cases {cases:10} | fcps {fcps:10.4f}")
47
48 for thr_id in range(192):
49       threadin.Thread(tarhet=worker, args=[thr_id]).start()
50
51 while threading.acite_count() > 0:
52       time.sleep(0.1)
```

Bibliografía

[1] Daniel K. Lee Ben L. Titzer y Jens Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. 2005. url: http://compilers.cs.ucla.edu/avrora/papers/avrora_ipsn2005.pdf (visitado 11-07-2021).

[2] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification. 1995. url: https://refspecs.linuxbase.org/elf/elf.pdf (visitado 09-07-2021).

[3] Khan Academy Dartmouth Computer Thomas Cormen y Devin Balkcom. Representar grafos. 2016. url: https://es.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs (visitado 11-07-2021).

[4] DragonJAR. Todo lo que debes saber sobre Radare2. 2021. url: https://book.rada.re/first_steps/overview.html (visitado 10-07-2021).

[5] Michael J. Eager Free Software Foundation. Introduction to the DWARF Debugging Format. 2012. url: http://dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf (visitado 10-07-2021).

[6] Several Authors Free Software Foundation. DWARF Debugging Information Format Version 5. 2017. url: http://dwarfstd.org/doc/DWARF5.pdf (visitado 11-07-2021).

[7] Alexey Frunze. StackOverflow assembly. 2013. url: https://stackoverflow.com/questions/15209993/what-does-opcode-ff350e204000-do?rq=1 (visitado 11-07-2021).

[8] Linux Fundation. ELF Header Linux Fundation. 2020. url: https://refspecs.linuxfoundation.org/elf/gabi4+/ch4.eheader.html (visitado 10-07-2021).

[9] GCC Gnu. GNU Basic Block. 2021. url: https://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html (visitado 10-07-2021).

[10] grapheverywhere. Qué son los grafos. 2021. url: https://www.grapheverywhere.com/que-son-los-grafos/ (visitado 11-07-2021).

[11] Diane Hosfel. Memory Safety. 2019. url: https://hacks.mozilla.org/2019/01/fearless-security-memory-safety/ (visitado 11-07-2021).

[12] IBM. XCOFF Object File Format. 2021. url: https://www.ibm.com/docs/en/aix/7.2?topic=formats-xcoff-object-file-format (visitado 10-07-2021).

[13] Open Source Software Initiative. GraphViz: graph Visualization Tool. 1999. url: https://graphviz.org/ (visitado 14-04-2021).

[14] ionos. Rust: el lenguaje de programación moderno. 2020. url: https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/rust-lenguaje-de-programacion/ (visitado 09-07-2021).

[15] Michael Kerrisk. elf(5) Linux manual page. 2021. url: https://man7.org/linux/man-pages/man5/elf.5.html (visitado 10-07-2021).

[16] Johannes Kinder. The Jakstab static analysis platform for binaries. 2012. url: http://www.jakstab.org/ (visitado 11-07-2021).

[17] lawebdelprogramado. Decompilador Definicion. 2021. url: https://www.lawebdelprogramador.com/diccionario/Descompilador/ (visitado 10-07-2021).

[18] John R. Levine. Object Files. 1999. url: https://archive.ph/20130125220014/http://www.iecc.com/linker/linker03.html (visitado 10-07-2021).

[19] lucidchart. Qué es un diagrama de flujo. 2020. url: https://www.lucidchart.com/pages/es/que-es-un-diagrama-de-flujo (visitado 11-07-2021).

[20] maijin pancake maijin y serveral authors. The Official Radare2 Book, Overview. 2021. url: https://book.rada.re/first_steps/overview.html (visitado 10-07-2021).

[21] Matt Milano. The Rust Programming Language: Its History and Why It Matters. 2020. url: https://www.talentopia.com/news/the-rust-programming-language-its-history-and-why/ (visitado 09-07-2021).

[22] palaksinghal9903. Basic Blocks in Compiler Design. 2021. url: https://www.geeksforgeeks.org/basic-blocks-in-compiler-design/ (visitado 10-07-2021).

[23] Santiago Urueña Pascual. Layout of an ELF file. 2007. url: https://commons.wikimedia.org/wiki/File:Elf-layout--en.svg (visitado 09-07-2021).

[24] RAE Real Academia Española. Definición grafo. 2021. url: https://dle.rae.es/grafo (visitado 11-07-2021).

[25] Unknow. DIAGRAMAS DE FLUJO DE CONTROL. 2021. url: https://docs.google.com/presentation/u/1/d/1OL4LOSnJ5Qoe6QyUeyZTcr6rjq_JEdrRN7T2i21a2y0/htmlpresent (visitado 10-07-2021).

[26] Unknown. Digrafos. 2012. url: http://tareasmd.blogspot.com/2012/07/digrafos.html (visitado 11-07-2021).

[27] Jeroen Ruigrok van der Werven. FreeBSD File Formats Manual ELF(5). 2020. url: https://www.freebsd.org/cgi/man.cgi?query=elf&sektion=5 (visitado 09-07-2021).

[28] Several Authors Wikipedia(EN). Assembly Language Wikipedia. 2021. url: https://en.wikipedia.org/wiki/Assembly_language (visitado 10-07-2021).

[29] Several Authors Wikipedia(EN). Bytecode Wikipedia. 2021. url: https://en.wikipedia.org/wiki/Bytecode (visitado 10-07-2021).

[30] Several Authors Wikipedia(EN). DOT: A graphical description language. 1999. url: https://en.wikipedia.org/wiki/DOT_(graph_description_language) (visitado 14-04-2021).

[31] Several Authors Wikipedia(EN). ELF Executable and Linkable Format. 2021. url: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format (visitado 09-07-2021).

[32] Several Authors Wikipedia(EN). Intermediate Representation Wikipedia. 2021. url: https://en.wikipedia.org/wiki/Intermediate_representation (visitado 10-07-2021).

[33] Several Authors Wikipedia(EN). Machin Code Wikipedia. 2021. url: https://en.wikipedia.org/wiki/Machine_code (visitado 10-07-2021).

[34] Several Authors Wikipedia(EN). Object Code Wikipedia. 2021. url: https://en.wikipedia.org/wiki/Object_code (visitado 10-07-2021).

[35] Several Authors Wikipedia(EN). Source Code Wikipedia. 2021. url: https://en.wikipedia.org/wiki/Source_code (visitado 10-07-2021).

[36] Several Authors Wikipedia(ES). Bloques Basicos Wikipedia. 2021. url: https://es.wikipedia.org/wiki/Bloque_b%C3%A1sico (visitado 21-04-2021).

[37] Several Authors Wikipedia(ES). Bytecode Wikipedia. 2021. url: https://es.wikipedia.org/wiki/Bytecode (visitado 10-07-2021).

[38] Several Authors Wikipedia(ES). Debuggin DWARF. 2021. url: https://es.wikipedia.org/wiki/DWARF (visitado 10-07-2021).

[39] Several Authors Wikipedia(ES). Decompilador Wikipedia. 2021. url: https://es.wikipedia.org/wiki/Decompilador (visitado 10-07-2021).

[40] Several Authors Wikipedia(ES). Desensamblador Wikipedia. 2021. url: https://es.wikipedia.org/wiki/Desensamblador (visitado 10-07-2021).

[41] Several Authors Wikipedia(ES). Ghidra, Wikipedia. 2021. url: https://en.wikipedia.org/wiki/Ghidra (visitado 11-07-2021).

[42] Several Authors Wikipedia(ES). Graphviz, Wikipedia. 2021. url: https://es.wikipedia.org/wiki/Graphviz (visitado 10-07-2021).

[43] Several Authors Wikipedia(ES). Interactive Disassembler (IDA), Wikipedia. 2019. url: https://es.wikipedia.org/wiki/Interactive_Disassembler (visitado 11-07-2021).

[44] Several Authors Wikipedia(ES). Intermediate Representation Wikipedia Español. 2021. url: https://es.wikipedia.org/wiki/Lenguaje_intermedio (visitado 10-07-2021).

[45] Several Authors Wikipedia(ES). Lenguaje Maquina Wikipedia. 2021. url: https://es.wikipedia.org/wiki/Lenguaje_de_m%C3%A1quina (visitado 10-07-2021).

[46] Several Authors Wikipedia(ES). Lista de Adyacencia, Wikipedia. 2021. url: https://es.wikipedia.org/wiki/Lista_de_adyacencia (visitado 10-07-2021).

[47] Several Authors Wikipedia(ES). Object Code Wikipedia. 2021. url: https://es.wikipedia.org/wiki/C%C3%B3digo_objeto (visitado 10-07-2021).

[48] Several Authors Wikipedia(ES). Relocatable Object Module Format(OMF). 2020. url: https://en.wikipedia.org/wiki/Relocatable_Object_Module_Format (visitado 10-07-2021).