UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

TFG del Grado en Ingeniería Informática

Simulador del microprocesador Intel 80386 Un enfoque orientado a ciberseguridad

Presentado por Miguel González Pampliega
en Universidad de Burgos — 7 de junio de 2026
Tutor: Pedro Sánchez Ortega

Índice general

Índice de figuras

Índice de tablas

Apéndice A

Plan de Proyecto Software

A.1. Introducción

A continuación, se va a proceder a detallar cuál ha sido la planificación seguida a la hora del desarrollo del proyecto. La aproximación de desarrollo de este proyecto ha sido una aproximación en cascada. Ha habido 4 grandes etapas en el desarrollo del proyecto de Software (véase Figura A.1).

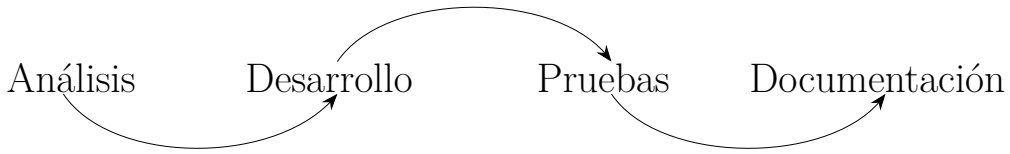

Figura A.1: Etapas planificadas para el desarrollo del proyecto

Cada una de estas fases contiene subtareas que contribuyen a la etapa a la que pertenecen. Al final de cada subtarea, se ha pretendido dejar el código en una forma consistente, es decir, sin errores manifiestos y con una pequeña documentación. Al final de cada una de estas subtareas, también se ha realizado un *commit* a algunos de los repositorios[1].

La idea principal de esta aproximación residía en ir introduciendo pequeñas mejoras en cuanto a usabilidad, e ir haciendo *commits* al repositorio de GitHub de manera recurrente, para llevar un registro de las versiones, y los cambios introducidos de una a otra.

[1]No solo existe un repositorio en este proyecto sino que algunas herramientas disponen de repositorio propio, ya que son opcionales a la hora de usar el simulador.

A la hora de realizar la planificación, debemos centrarnos tanto en la planificación temporal, como en el estudio de la viabilidad .

A.2. Planificación temporal

Cabe resaltar que el grueso del desarrollo del proyecto, que es el motor de simulación, conformado por la implementación de las estructuras de datos y la lógica de la ejecución de las instrucciones, ha sido desarrollado de manera anterior a la aprobación del proyecto por parte del Tribunal. Esto se debe a la complejidad del proyecto y al tiempo necesario para llevarlo a cabo.
Entender cómo funciona el procesador a nivel de programador ya lleva un número consistente de horas de por sí, pero entender la lógica detrás de cada instrucción (que son aproximadamente 205), y luego transformar esa lógica a las estructuras de datos simuladas, es un arduo trabajo que lleva varios meses.

Esto puede verse en el fichero de lenguaje C que implementa las instrucciones, pues tiene una longitud aproximada de 8000 líneas, y no es la única parte a desarrollar. Para poder realizar pruebas sobre toda esta carga de trabajo ha sido necesario crear nuestros propios *tests* automáticos que comprueben todas las instrucciones y reflejen en un informe si ha ocurrido algún fallo, tal como se puede ver en el Apartado G.1.

Se puede ver con detalle en la Figura A.2 y en la Figura A.3 los diagramas de *Gantt* que detalla la planificación seguida durante el proyecto. Tanto la planificación inicial realizada antes de comenzar el proyecto, como la planificación real seguida, con sus retrasos y sus imprevistos. Estos diagramas ha sido realizado siguiendo la metodología propuesta en [1].

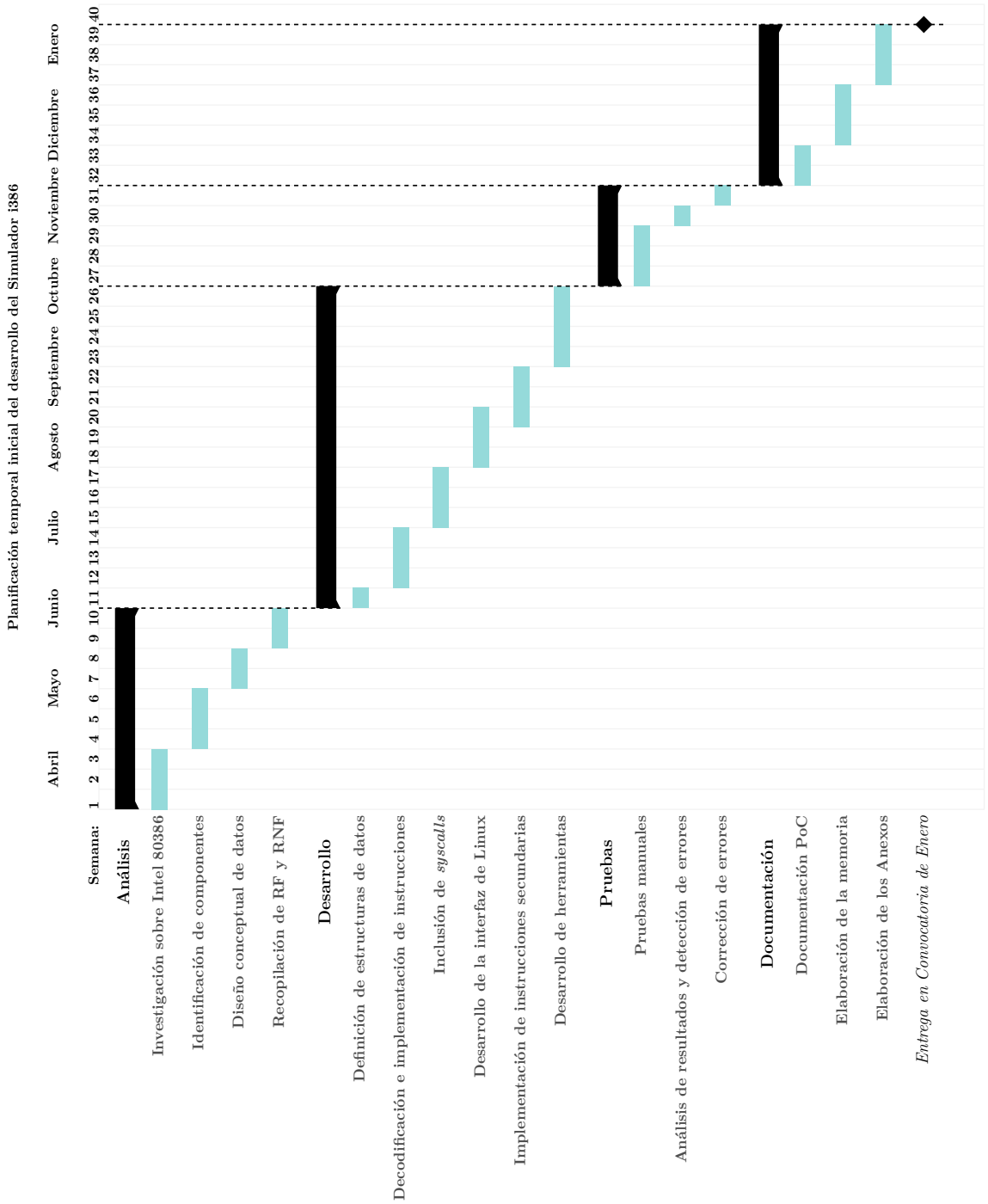

Figura A.2: Diagrama de Gantt de la planificación inicial del proyecto

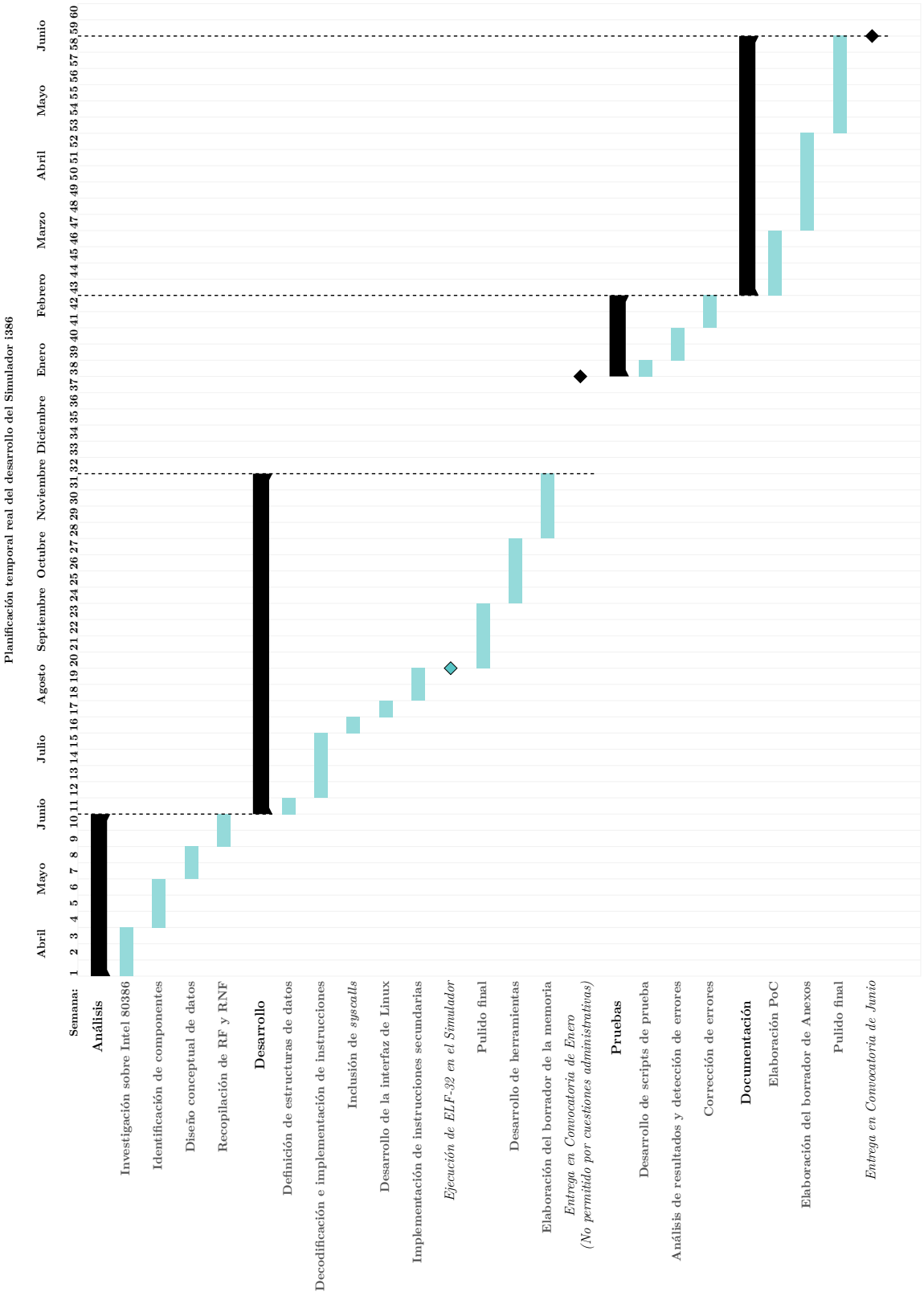

Figura A.3: Diagrama de Gantt de la planificación real del proyecto

A.3. Estudio de viabilidad

Antes de llevar a cabo el proyecto, fue necesario evaluar la viabilidad de este, en distintos ámbitos.

Viabilidad económica

La viabilidad económica debe ser computada en función de las horas de trabajo que han sido necesarias para la concepción del simulador. Por ello, primero debemos determinar la cantidad de horas, y luego aplicar el cálculo del precio de cada hora. Podemos ver la distribución de horas y costes en la Tabla A.1.

Tarea	**Nº de horas**	**Precio de la hora**	**Total**
Análisis	40 horas	10 €	400 €
Diseño	60 horas	12 €	720 €
Codificación	250 horas	15 €	3750 €
Pruebas	30 horas	8 €	240 €
Documentación	120 horas	11 €	1320 €
		Coste total del proyecto	6430 €

Tabla A.1: Coste del desarrollo del proyecto

Adicionalmente, conllevaría un desembolso económico si quisiéramos desplegar una página web que informara acerca de los beneficios de nuestro simulador, y que contuviera información acerca del código y dónde puede ser encontrado. El presupuesto para desarrollar una página web de este estilo se puede ver en la Tabla A.2. El presupuesto anual para mantener una página web de este tipo se puede ver en la Tabla A.3.

Tarea	**Nº de horas**	**Precio de la hora**	**Coste**
Redacción de contenido	5 horas	15 €	120 €
Aplicación de estilos	20 horas	20 €	400 €
Coste total			520 €

Tabla A.2: Coste del desarrollo de la web

Servicio	Proveedor	Coste anual
Nombre de dominio	PiensaSolutions	15 €
Servidor Web	PiensaSolutions	12 €
Certificado SSL	Let's Encrypt	0 €
Coste total anual		27 €

Tabla A.3: Coste del despliegue de la web

De igual manera tenemos que tener en cuenta la amortización de los equipos utilizados para el desarrollo del proyecto. El proyecto ha sido desarrollado en dos equipos, un equipo de sobremesa de unos 1300 € y cuya vida útil se estima en 10 años, por lo que la amortización de un proyecto de 15 meses sería de 163 €. El otro equipo es un portátil de 1000 € cuya vida útil se estima en 7 años, por lo que la amortización de 15 meses sería de 179 €.

También podríamos incluir el gasto de un **certificado digital de firma de código**, que consiste en un certificado que permite firmar nuestro software. Son expedidos por entidades certificadoras reconocidas y sirven para que los sistemas operativos no detecten como *malware* el software al instalarlo y tengan trazabilidad hasta la empresa que lo ha desarrollado. Este certificado ronda los 200 dólares estadounidenses al año.

Viabilidad funcional

Debido a la complejidad del proyecto, tanto por su dificultad de abstracción como por la dificultad de codificación, fue necesario llevar a cabo un estudio previo de viabilidad.

En primer lugar, se debía escoger cuidadosamente un lenguaje de programación que permitiera mantener el control total sobre las estructuras de datos que utiliza el programa, como pueden ser las variables y la memoria reservada en el *heap*. De manera adicional, debía ser un lenguaje de programación que mantuviera una relación estrecha con el Kernel del sistema operativo, por lo que se eligió el lenguaje C.

En segundo lugar, existía complejidad oculta acerca del espacio de direcciones del simulador, ya que requería de 2^{32} bytes, que corresponden a 4 GB de memoria para un único programa, lo que podría sonar excesivo y un mal uso de los recursos del equipo. Sin embargo, esta problemática es resuelta fácilmente mediante el siguiente mecanismo. Cuando se inicia el simulador, el programa reserva 2^{32} bytes de memoria en el heap, pero esos 4GB reservados no son memoria en sí, sino direcciones de memoria virtual.

Por lo que aunque realmente hayamos reservado 4.294.967.296 direcciones de memoria, solo repercutirá en consumo real de la máquina host, las direcciones de memoria virtual a la que hayamos accedido para lectura o escritura. Es decir, aunque reservemos 4.000 millones de direcciones de memoria del heap, si sólo accedemos a 100 de ellas, el consumo de memoria física del programa será de solo 100 Bytes. El consumo de direcciones de memoria virtual no es un problema ya que los ordenadores modernos tienen un espacio de direcciones de 2^{64} bytes, que correspondería a ejecutar el simulador 4.000 millones de veces.

En tercer lugar, existía una complejidad real en la replicación de las estructuras de datos del microprocesador real como estructuras de datos de un programa C, como pueden ser los registros o la GDT (*Global Descriptors Table*). Además de cómo las instrucciones deberían afectar a dichas estructuras de datos simuladas. Finalmente, se encontró una manera de implementar las instrucciones para que afectaran de manera verosímil a dichas estructuras de datos.

Viabilidad legal

La viabilidad legal, aunque no es un problema real, es un aspecto importante a tener en cuenta. Esto se debe a que cada librería, framework y programa utilizado durante el desarrollo del simulador propuesto en este trabajo, tiene una licencia distinta, y debemos tener en cuenta las cláusulas de estas licencias. Dichas licencias son:

Capstone licencia BSD de 3 cláusulas.

NASM licencia BSD simplificada.

.NET / Windows Forms licencia MIT.

Por ello, si juntamos todas las obligaciones impuestas por las cláusulas de estas licencias, obtenemos lo siguiente.

- Incluir las licencias de las librerías utilizadas en el proyecto.
- Mantener los avisos de *copyright*.
- No usar el nombre de los autores sin su consentimiento (Según la BSD de 3 cláusulas).

Sin embargo, estas licencias permiten realizar distribuciones comerciales, y modificar el código sin necesidad de publicar los cambios.

Apéndice B

Especificación de Requisitos

B.1. Introducción

Los casos de uso representan las posibles interacciones con el sistema desde el punto de vista del usuario. En este apartado, veremos los casos de uso de nuestro simulador del i386, y como influyen en ellos los requisitos funcionales. Si bien los requisitos funcionales son los mismos para la versión de Linux y la de Windows, los casos de uso varían ligeramente, ya que algunas funcionalidades no están disponibles en Windows, y por lo tanto los usuarios no pueden interactuar con ellas como lo harían en Windows.

B.2. Casos de uso de la interfaz de Linux

CU-1	**Seleccionar las opciones de ejecución del simulador (argumentos de línea de comandos).**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-5.1, RF-5.2
Descripción	Consiste en que el usuario seleccione una serie de opciones, habiendo un par de ellas obligatorias (como el modo de la interfaz y el formato del fichero) y otras opciones que aportan funcionalidades como NX, ASLR o rango de pila personalizado.
Precondición	No hay un fichero ejecutable ya cargado en memoria.
Acciones	1. Comprobar que no existan incompatibilidades entre argumentos. 2. Procesar el modo de ejecución (Interfaz o normal). 3. Procesar el formato de fichero ejecutable (ELF o RAW). 4. Procesar el resto de argumentos. 5. Almacenar las opciones indicadas.
Postcondición	Las opciones han sido almacenadas en memoria para tenerse en cuenta cuando se cargue el programa. De haber elegido un rango de pila personalizado, o aleatorio (ASLR), se establece dicho rango de pila. De modo contrario, se establece el rango de pila por defecto.
Excepciones	Dos o más argumentos incompatibles. Faltan opciones obligatorias. Se muestra la ayuda y se finaliza el programa.
Importancia	Alta.

Tabla B.1: CU-1 Seleccionar opciones de ejecución del simulador.

CU-2	**Cargar archivo ejecutable en memoria.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-1.1, RF-3.1, RF-5.1, RF-5.2
Descripción	Consiste en que el usuario seleccione un archivo ejecutable, en los formatos soportados (RAW o ELF-32), para que el programa lo cargue en memoria, mostrando las instrucciones que incluye.
Precondición	Existe un fichero ejecutable de los formatos soportados, compilado estáticamente y con *No-PIE*. Las opciones del simulador ya han sido seleccionadas. La memoria ha sido inicializada.
Acciones	1. Seleccionar archivo ejecutable. 2. Leer cabeceras y procesarlas (según formato). 3. Cargar sección ejecutable en memoria. 4. Desensamblar las instrucciones de la sección ejecutable y almacenarlas. 5. Si el archivo es ELF-32 y el modo es interfaz, leer los nombres de las funciones y sus direcciones de memoria, y almacenarlas para su uso.
Postcondición	El programa queda cargado en memoria, la pila y registros inicializados, y el registro EIP apunta a la instrucción de inicio del programa.
Excepciones	Fichero ejecutable no válido. Se muestra el código de error y finaliza el programa.
Importancia	Alta.

Tabla B.2: CU-2 Cargar archivo ejecutable en memoria.

CU-3	**Step. Ejecutar una instrucción.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-1.1, RF-1.2, RF-3.1, RF-4, RF-6.2.1
Descripción	Consiste en que el simulador ejecute la instrucción siguiente. Esta instrucción es apuntada por el registro EIP.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Han sido inicializada la pila y todos los registros, en especial el EIP y el ESP.
Acciones	1. Obtener la dirección de memoria apuntada por el registro EIP (Fetch). 2. Decodificar la instrucción y sus argumentos a partir del bytecode que inicia en dicha dirección de memoria (Decode). 3. Ejecutar la instrucción (Execute). 4. Acceso a memoria si es necesario (Memory). 5. Guardar el resultado (si lo hubiera) en el registro correspondiente (Write-Back).
Postcondición	El simulador ha actualizado sus estructuras de datos (registros y memoria) en función de la instrucción ejecutada, y ahora EIP apunta a la siguiente.
Excepciones	Instrucción devuelve el valor especial `0xdeadbeef`, que indica que se debe finalizar el programa.
Importancia	Media.

Tabla B.3: CU-3 Ejecutar una instrucción.

CU-4	**Continue. Ejecutar instrucciones hasta alcanzar un *breakpoint*.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-1.1, RF-1.2, RF-3.1, RF-4, RF-6.2.2, RF-6.2.4
Descripción	Consiste en que el simulador ejecute tantas instrucciones como sea necesario (incluso hasta la finalización del programa) hasta que encuentre una dirección de memoria que corresponda a un *breakpoint*.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la tabla de *breakpoints*, así como sus variables (contador, tamaño máximo...). Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP.
Acciones	1. Obtener la dirección de memoria apuntada por el registro EIP (Fetch). 2. Si la dirección de memoria está contenida en la tabla de *breakpoints*, abortar operación. En el caso contrario, seguir ejecutando. 3. Decodificar la instrucción y sus argumentos a partir del bytecode que inicia en dicha dirección de memoria (Decode). 4. Ejecutar la instrucción (Execute). 5. Acceso a memoria si es necesario (Memory). 6. Guardar el resultado (si lo hubiera) en el registro correspondiente (Write-Back). 7. Actualizar el registro EIP. 8. Saltar al paso 1.
Postcondición	El simulador ha actualizado sus estructuras de datos (registros y memoria) en función de las instrucciones ejecutadas, y ahora EIP apunta a la siguiente, que corresponde a un punto de parada.
Excepciones	Instrucción devuelve el valor especial `0xdeadbeef`, que indica que se debe finalizar el programa.
Importancia	Media.

Tabla B.4: CU-4 Ejecutar instrucciones hasta alcanzar un *breakpoint*.

CU-5	**No-Enter. Ejecutar instrucciones hasta alcanzar la instrucción siguiente.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-1.1, RF-1.2, RF-3.1, RF-4, RF-6.2.3
Descripción	Consiste en que el simulador ejecute tantas instrucciones como sea necesario (incluso hasta la finalización del programa) hasta que encuentre la dirección de memoria inmediatamente siguiente a la instrucción inicial. Tiene como objetivo permitir evitar entrar en llamadas a funciones (CALL) o bucles que usen saltos.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP.
Acciones	1. Obtener la dirección de memoria de parada, que corresponde al EIP actual sumando el tamaño de la instrucción apuntada por el mismo. Es decir, la dirección de la instrucción siguiente. 2. Obtener la dirección de memoria apuntada por el registro EIP (Fetch). 3. Si la dirección de memoria de parada es igual a la del EIP, abortar operación. En el caso contrario, seguir ejecutando. 4. Decodificar la instrucción y sus argumentos a partir del bytecode que inicia en dicha dirección de memoria (Decode). 5. Ejecutar la instrucción (Execute). 6. Acceso a memoria si es necesario (Memory). 7. Guardar el resultado (si lo hubiera) en el registro correspondiente (Write-Back). 8. Actualizar el registro EIP. 9. Saltar al paso 2.
Postcondición	El simulador ha actualizado sus estructuras de datos (registros y memoria) en función de las instrucciones ejecutadas, y ahora EIP apunta a la siguiente, que corresponde a la instrucción inmediatamente siguiente de la que hemos empezado.
Excepciones	Instrucción devuelve el valor especial `0xdeadbeef`, que indica que se debe finalizar el programa.
Importancia	Media.

Tabla B.5: CU-5 Ejecutar instrucciones hasta alcanzar la siguiente.

CU-6	**Breakpoint. Añadir punto de parada.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-6.2.4
Descripción	Consiste en añadir un punto de parada en una dirección de memoria. Si se ejecuta la opción *Continue*, se detendrá al llegar a alguno de estos puntos de parada.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP. Ha sido inicializada la tabla de *breakpoints*, así como sus variables (contador, tamaño máximo...)
Acciones	1. El usuario indica la dirección de memoria en la que quiere establecer el punto de parada. 2. Se comprueba el formato de la entrada del usuario. 3. Si el formato es erróneo, salta al paso 1. 4. Se añade el punto de parada a la tabla de *breakpoints*. 5. Se incrementa el contador de *breakpoints* y se comprueba que no haya *overflow* (el contador no puede ser mayor que el número máximo de puntos de parada).
Postcondición	Nuevo punto de parada añadido y contador incrementado. Si al incrementar el contador hay *overflow*, se reinicia a 0. Se muestra el índice del punto de parada al usuario.
Excepciones	Ninguna.
Importancia	Media.

Tabla B.6: CU-6 Añadir punto de parada.

CU-7	**Delete. Eliminar punto de parada.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-6.2.5
Descripción	Consiste en eliminar un punto de parada basado en su índice. Se accede a la tabla de *breakpoints* y se establece el valor a 0, borrando la dirección de memoria que hubiera anteriormente.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP. Ha sido inicializada la tabla de *breakpoints*, así como sus variables (contador, tamaño máximo...)
Acciones	1. El usuario indica el índice del punto de parada a eliminar. 2. Se comprueba el formato de la entrada del usuario. 3. Si el formato es erróneo, salta al paso 1. 4. Se comprueba que el valor está entre 0 y el número máximo de puntos de parada, para evitar accesos a memoria maliciosos. Se realiza la operación módulo con el índice otorgado por el usuario y el valor máximo, para asegurar su pertenencia al rango válido. 5. Se accede a la entrada de la tabla de dicho índice, y se establece a 0, eliminando la dirección existente previamente.
Postcondición	Punto de parada eliminado.
Excepciones	Ninguna.
Importancia	Baja.

Tabla B.7: CU-7 Eliminar punto de parada.

CU-8	**Show Hex. Mostrar el contenido hexadecimal de una dirección de memoria.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-6.2.7
Descripción	Consiste en mostrar el contenido hexadecimal de una dirección de memoria, mostrando los 4 bytes en *Little-Endian* que le corresponden.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP.
Acciones	1. El usuario indica la dirección de memoria cuyo contenido quiere consultar. 2. Se comprueba el formato de la entrada del usuario. 3. Si el formato es erróneo, salta al paso 1. 4. Se accede al contenido de la dirección de memoria. 5. Se muestra al usuario el contenido hexadecimal de dicha dirección de memoria.
Postcondición	Ninguna.
Excepciones	Ninguna.
Importancia	Baja.

Tabla B.8: CU-8 Mostrar el contenido hexadecimal de una dirección de memoria.

CU-9	**Show String. Mostrar la cadena que comienza en una dirección de memoria.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-6.2.8
Descripción	Consiste en mostrar el contenido en forma de cadena de caracteres de una dirección de memoria, mostrando tantos bytes como sean necesarios hasta alcanzar un byte nulo `0x00`, que es el finalizador de cadenas.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP.
Acciones	1. El usuario indica la dirección de memoria cuyo contenido quiere consultar. 2. Se comprueba el formato de la entrada del usuario. 3. Si el formato es erróneo, salta al paso 1. 4. Se accede al contenido de la dirección de memoria. 5. Se muestra al usuario el contenido de dicha dirección de memoria en forma de cadena.
Postcondición	Ninguna.
Excepciones	Ninguna.
Importancia	Baja.

Tabla B.9: CU-9 Mostrar la cadena que comienza en una dirección de memoria.

CU-10	**Cambiar enfoque de las ventanas de la interfaz.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-3.1, RF-3.2
Descripción	Consiste en seleccionar la ventana con el enfoque activo, que la convierte en la afectada de ciertas operaciones, como el *Scroll* o *Find*.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP.
Acciones	1. El usuario indica la ventana sobre la que aplicar el enfoque. 2. Se cambia el enfoque a la ventana indicada.
Postcondición	Enfoque cambiado a la ventana indicada.
Excepciones	Ninguna.
Importancia	Baja.

Tabla B.10: CU-10 Cambiar enfoque de las ventanas de la interfaz.

CU-11	**Find. Encontrar una dirección de memoria.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-3.1, RF-3.2, RF-6.2.6
Descripción	Consiste en buscar una dirección de memoria en la ventana con el enfoque activo, y si es encontrada, establecerla en la parte de arriba de dicha ventana. Puede ser una dirección asociada a una instrucción en la ventana de código, o una dirección asociada a un valor en pila en la ventana de la pila.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP. Enfoque por defecto en el código.
Acciones	1. El usuario indica la dirección de memoria que quiere encontrar y mostrar. 2. Se comprueba el formato de la entrada del usuario. 3. Si el formato es erróneo, salta al paso 1. 4. Se busca la dirección de memoria. 5. Si es encontrada, se muestra la primera en su respectiva ventana.
Postcondición	Cambio en la información mostrada en la interfaz.
Excepciones	Ninguna.
Importancia	Baja.

Tabla B.11: CU-11 Encontrar una dirección de memoria.

CU-12	**Scroll. Desplazarse por las ventanas de la interfaz.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-3.1, RF-3.2
Descripción	Consiste en desplazarse una posición hacia arriba o hacia abajo en la ventana de la interfaz con el enfoque activo.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP. Enfoque por defecto en el código.
Acciones	1. El usuario indica la dirección en la que quiere desplazarse, ascendente o descendente. 2. La ventana con el enfoque activo es afectada en función de la elección del usuario, desplazando el contenido hacia arriba o hacia abajo.
Postcondición	Cambio en la información mostrada en la interfaz.
Excepciones	Ninguna.
Importancia	Baja.

Tabla B.12: CU-12 Desplazarse por las ventanas de la interfaz.

B.3. Casos de uso de la interfaz de Windows

CU-1	**Cargar archivo ejecutable en memoria.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-1.1, RF-3.1, RF-5.2
Descripción	Consiste en que el usuario seleccione un archivo ejecutable, en los formatos soportados (RAW o ASM), para que el programa lo cargue en memoria, mostrando las instrucciones que incluye.
Precondición	Existe un fichero ejecutable de los formatos soportados. Si el fichero es ASM, debe tener una sintaxis correcta para no dar errores de compilación. La memoria ha sido inicializada.
Acciones	1. Seleccionar el tipo de archivo ejecutable. 2. Si el formato es RAW, cargarlo íntegro en memoria. Si es ASM, compilar dicho ASM como un fichero ejecutable binario temporal con NASM para cargarlo como fichero RAW. 3. Desensamblar las instrucciones de la sección ejecutable y almacenarlas. 4. Inicializar el EIP a 0.
Postcondición	El programa queda cargado en memoria, la pila y registros inicializados, y el registro EIP apunta a la instrucción de inicio del programa.
Excepciones	Error al compilar el fichero ASM. Se muestra el código de error y finaliza el programa.
Importancia	Alta.

Tabla B.13: CU-1 Cargar archivo ejecutable en memoria.

CU-2	**Step. Ejecutar una instrucción.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-1.1, RF-1.2, RF-3.1, RF-6.2.1
Descripción	Consiste en que el simulador ejecute la instrucción siguiente. Esta instrucción es apuntada por el registro EIP.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Han sido inicializada la pila y todos los registros, en especial el EIP y el ESP.
Acciones	1. Obtener la dirección de memoria apuntada por el registro EIP (Fetch). 2. Decodificar la instrucción y sus argumentos a partir del bytecode que inicia en dicha dirección de memoria (Decode). 3. Ejecutar la instrucción (Execute). 4. Acceso a memoria si es necesario (Memory). 5. Guardar el resultado (si lo hubiera) en el registro correspondiente (Write-Back).
Postcondición	El simulador ha actualizado sus estructuras de datos (registros y memoria) en función de la instrucción ejecutada, y ahora EIP apunta a la siguiente.
Excepciones	Instrucción devuelve el valor especial `0xf4`, que indica que se debe finalizar el programa.
Importancia	Media.

Tabla B.14: CU-2 Ejecutar una instrucción.

CU-3	**Continue. Ejecutar instrucciones hasta alcanzar un *breakpoint*.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-1.1, RF-1.2, RF-3.1, RF-6.2.2, RF-6.2.4
Descripción	Consiste en que el simulador ejecute tantas instrucciones como sea necesario (incluso hasta la finalización del programa) hasta que encuentre una dirección de memoria que corresponda a un *breakpoint*.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la tabla de *breakpoints*, así como sus variables (contador, tamaño máximo...). Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP.
Acciones	1. Obtener la dirección de memoria apuntada por el registro EIP (Fetch). 2. Si la dirección de memoria está contenida en la tabla de *breakpoints*, abortar operación. En el caso contrario, seguir ejecutando. 3. Decodificar la instrucción y sus argumentos a partir del bytecode que inicia en dicha dirección de memoria (Decode). 4. Ejecutar la instrucción (Execute). 5. Acceso a memoria si es necesario (Memory). 6. Guardar el resultado (si lo hubiera) en el registro correspondiente (Write-Back). 7. Actualizar el registro EIP. 8. Saltar al paso 1.
Postcondición	El simulador ha actualizado sus estructuras de datos (registros y memoria) en función de las instrucciones ejecutadas, y ahora EIP apunta a la siguiente, que corresponde a un punto de parada.
Excepciones	Instrucción devuelve el valor especial `0xf4`, que indica que se debe finalizar el programa.
Importancia	Media.

Tabla B.15: CU-3 Ejecutar instrucciones hasta alcanzar un *breakpoint*.

CU-4	**No-Enter. Ejecutar instrucciones hasta alcanzar la instrucción siguiente.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-1.1, RF-1.2, RF-3.1, RF-6.2.3
Descripción	Consiste en que el simulador ejecute tantas instrucciones como sea necesario (incluso hasta la finalización del programa) hasta que encuentre la dirección de memoria inmediatamente siguiente a la instrucción inicial. Tiene como objetivo permitir evitar entrar en llamadas a funciones (CALL) o bucles que usen saltos.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP.
Acciones	1. Obtener la dirección de memoria de parada, que corresponde al EIP actual sumando el tamaño de la instrucción apuntada por el mismo. Es decir, la dirección de la instrucción siguiente. 2. Obtener la dirección de memoria apuntada por el registro EIP (Fetch). 3. Si la dirección de memoria de parada es igual a la del EIP, abortar operación. En el caso contrario, seguir ejecutando. 4. Decodificar la instrucción y sus argumentos a partir del bytecode que inicia en dicha dirección de memoria (Decode). 5. Ejecutar la instrucción (Execute). 6. Acceso a memoria si es necesario (Memory). 7. Guardar el resultado (si lo hubiera) en el registro correspondiente (Write-Back). 8. Actualizar el registro EIP. 9. Saltar al paso 2.
Postcondición	El simulador ha actualizado sus estructuras de datos (registros y memoria) en función de las instrucciones ejecutadas, y ahora EIP apunta a la siguiente, que corresponde a la instrucción inmediatamente siguiente de la que hemos empezado.
Excepciones	Instrucción devuelve el valor especial `0xf4`, que indica que se debe finalizar el programa.
Importancia	Media.

Tabla B.16: CU-4 Ejecutar instrucciones hasta alcanzar la siguiente.

CU-5	**Breakpoint. Añadir punto de parada.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-6.2.4
Descripción	Consiste en añadir un punto de parada en una dirección de memoria. Si se ejecuta la opción *Continue*, se detendrá al llegar a alguno de estos puntos de parada.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP. Ha sido inicializada la tabla de *breakpoints*, así como sus variables (contador, tamaño máximo...)
Acciones	1. El usuario indica la dirección de memoria en la que quiere establecer el punto de parada. 2. Se comprueba el formato de la entrada del usuario. 3. Si el formato es erróneo, salta al paso 1. 4. Se añade el punto de parada a la tabla de *breakpoints*. 5. Se incrementa el contador de *breakpoints* y se comprueba que no haya *overflow* (el contador no puede ser mayor que el número máximo de puntos de parada).
Postcondición	Nuevo punto de parada añadido y contador incrementado. Si al incrementar el contador hay *overflow*, se reinicia a 0. Se muestra el índice del punto de parada al usuario.
Excepciones	Ninguna.
Importancia	Media.

Tabla B.17: CU-5 Añadir punto de parada.

CU-6	**Delete. Eliminar punto de parada.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-6.2.5
Descripción	Consiste en eliminar un punto de parada basado en su índice. Se accede a la tabla de *breakpoints* y se establece el valor a 0, borrando la dirección de memoria que hubiera anteriormente.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP. Ha sido inicializada la tabla de *breakpoints*, así como sus variables (contador, tamaño máximo...)
Acciones	1. El usuario indica el índice del punto de parada a eliminar. 2. Se comprueba el formato de la entrada del usuario. 3. Si el formato es erróneo, salta al paso 1. 4. Se comprueba que el valor está entre 0 y el número máximo de puntos de parada, para evitar accesos a memoria maliciosos. Se realiza la operación módulo con el índice otorgado por el usuario y el valor máximo, para asegurar su pertenencia al rango válido. 5. Se accede a la entrada de la tabla de dicho índice, y se establece a 0, eliminando la dirección existente previamente.
Postcondición	Punto de parada eliminado.
Excepciones	Ninguna.
Importancia	Baja.

Tabla B.18: CU-6 Eliminar punto de parada.

CU-7	**Show Hex. Mostrar el contenido hexadecimal de una dirección de memoria.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-6.2.7
Descripción	Consiste en mostrar el contenido hexadecimal de una dirección de memoria, mostrando los 4 bytes en *Little-Endian* que le corresponden.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP.
Acciones	1. El usuario indica la dirección de memoria cuyo contenido quiere consultar. 2. Se comprueba el formato de la entrada del usuario. 3. Si el formato es erróneo, salta al paso 1. 4. Se accede al contenido de la dirección de memoria. 5. Se muestra al usuario el contenido hexadecimal de dicha dirección de memoria.
Postcondición	Ninguna.
Excepciones	Ninguna.
Importancia	Baja.

Tabla B.19: CU-7 Mostrar el contenido hexadecimal de una dirección de memoria.

CU-8	**Show String. Mostrar la cadena que comienza en una dirección de memoria.**
Versión	1.0
Autor	Miguel González Pampliega
Requisitos asociados	RF-6.2.8
Descripción	Consiste en mostrar el contenido en forma de cadena de caracteres de una dirección de memoria, mostrando tantos bytes como sean necesarios hasta alcanzar un byte nulo `0x00`, que es el finalizador de cadenas.
Precondición	La memoria ha sido inicializada y el programa cargado en memoria. Ejecución en modo interfaz. Ha sido inicializada la pila y todos los registros, en especial el EIP y el ESP.
Acciones	1. El usuario indica la dirección de memoria cuyo contenido quiere consultar. 2. Se comprueba el formato de la entrada del usuario. 3. Si el formato es erróneo, salta al paso 1. 4. Se accede al contenido de la dirección de memoria. 5. Se muestra al usuario el contenido de dicha dirección de memoria en forma de cadena.
Postcondición	Ninguna.
Excepciones	Ninguna.
Importancia	Baja.

Tabla B.20: CU-8 Mostrar la cadena que comienza en una dirección de memoria.

B.4. Objetivos generales

El objetivo principal, que veremos reflejados en los requisitos funcionales y no funcionales, es que el simulador a desarrollar sea una herramienta de docencia, y sea utilizado como material docente en entornos universitarios. Por ello, las funcionalidades descritas por los requisitos funcionales deben tratar de ser lo más sencillas posible y de igual manera, hacer más sencilla la labor del docente a la hora de explicar los contenidos asociados al simulador,

como pueden ser los registros, pila, o direcciones de memoria. Las funcionalidades descritas por estos requisitos funcionales también deben hacer más sencilla la labor del estudiante a la hora de seguir los contenidos, y permitir *debuggear* los programas que se ejecuten en el simulador.

Asimismo, los requisitos no funcionales tienen como objetivo describir otro tipo de propiedades, ya no funcionales, sino visuales, que permitan que se entienda el contenido del simulador de un plumazo. Para ello se pueden utilizar herramientas de UI, colores y otras alternativas que permitan que tanto docente como alumno puedan entender lo que están viendo.

B.5. Catálogo de requisitos

A continuación, se va a proceder a mencionar todos los requisitos existentes, tanto funcionales como no funcionales. En esta sección serán nombrados ordenadamente, mientras que serán explicados con detalle en la siguiente sección.

Requisitos funcionales

A continuación se muestra el catálogo de Requisitos Funcionales (RF).

RF 1 Instrucciones

- **RF 1.1** Desensamblar instrucciones
  - **RF 1.1.1** Nemotécnico
  - **RF 1.1.2** Argumentos
- **RF 1.2** Ejecutar instrucciones
  - **RF 1.2.1** Actualizar registros
  - **RF 1.2.2** Actualizar memoria
  - **RF 1.2.3** Actualizar banderas

RF 2 Registros

- **RF 2.1** Acumulador
- **RF 2.2** Propósito general
- **RF 2.3** Propósito específico
- **RF 2.4** Selectores de segmento
- **RF 2.5** Banderas

RF 3 Memoria

- **RF 3.1** Memoria de código
- **RF 3.2** Pila
- **RF 3.3** Segmentación

RF 4 Llamadas al sistema

- **RF 4.1** Implementación
- **RF 4.2** Traducción
- **RF 4.3** Argumentos

RF 5 Manejo de ficheros

- **RF 5.1** ELF-32
  - **RF 5.1.1** Loader
  - **RF 5.1.2** Nombres de funciones
  - **RF 5.1.3** Obtener EIP
  - **RF 5.1.4** Preparar la pila
- **RF 5.2** Raw
  - **RF 5.2.1** Loader
  - **RF 5.2.2** Obtener EIP

RF 6 Interfaz

- **RF 6.1** Ventanas
  - **RF 6.1.1** Registros
  - **RF 6.1.2** Código
  - **RF 6.1.3** Pila
  - **RF 6.1.4** Terminal
- **RF 6.2** Comandos
  - **RF 6.2.1** Step
  - **RF 6.2.2** Continue
  - **RF 6.2.3** No Enter
  - **RF 6.2.4** Breakpoint
  - **RF 6.2.5** Delete
  - **RF 6.2.6** Find
  - **RF 6.2.7** Show Hex
  - **RF 6.2.8** Show String

Requisitos No Funcionales

A continuación se muestra el catálogo de Requisitos No Funcionales (RNF).

RNF 1 Separar ventanas de la interfaz

RNF 2 Resaltar EIP

RNF 3 Resaltar registros modificados

RNF 4 Colorear direcciones de memoria

RNF 5 Colorear funciones

B.6. Especificación de requisitos

En este apartado, se van a explicar los requisitos funcionales y no funcionales mencionados anteriormente, con la finalidad de entender profundamente a qué se refiere cada uno, en vez de mostrar una frase esquemática.

Requisitos Funcionales

A continuación se muestra la especificación de Requisitos Funcionales (RF).

RF 1 **Instrucciones** se deben poder manejar el programa instrucción a instrucción.

- **RF 1.1** **Desensamblar instrucciones** se deben poder desensamblar instrucciones a partir del *bytecode*.
  - **RF 1.1.1** **Nemotécnico** se debe poder extraer el nemotécnico de la instrucción al realizar su desensamblaje.
  - **RF 1.1.2** **Argumentos** se debe poder extraer el número de argumentos y su contenido al realizar el desensamblaje de la instrucción.
- **RF 1.2** **Ejecutar instrucciones** se deben poder ejecutar las instrucciones
  - **RF 1.2.1** **Actualizar registros** se deben actualizar los registros (en especial el EIP) después de la ejecución de cada instrucción.

RF 1.2.2 Actualizar memoria se deben actualizar las direcciones de memoria afectadas después de la ejecución de cada instrucción.

RF 1.2.3 Actualizar banderas se deben actualizar las banderas después de la ejecución de cada instrucción.

RF 2 Registros se deben simular los registros del microprocesador, preferiblemente con estructuras de datos de 32 bits.

RF 2.1 Acumulador se debe implementar el registro EAX, que funciona como acumulador en gran parte de las operaciones aritméticas. También se deben poder utilizar los subregistros del EAX, que son AX, AH y AL.

RF 2.2 Propósito general se deben implementar los registros de propósito general EBX, ECX y EDX, pudiendo usar también sus subregistros (BX, CX, DX, BH, CH, DH, ...).

RF 2.3 Propósito específico se deben implementar los registros de propósito específico, como el EBP, el ESP atendiendo a las operaciones de la pila (PUSH, POP, RET...) y modificarlo en consecuencia, y el EIP, modificándolo en consecuencia cada vez que se ejecuta una instrucción (ya sea una instrucción normal o de control de flujo como CALL, RET o JMP).

RF 2.4 Selectores de segmento se deben implementar los selectores de segmento de manera funcional, pero con base 0 y límite 0xFFFFFF, para conseguir una segmentación de memoria FLAT.

RF 2.5 Banderas se debe implementar el registro EFLAGS, así como las funcionalidades necesarias para testear y modificar las banderas contenidas en dicho registro.

RF 3 Memoria se debe simular la memoria del microprocesador, con un espacio de direcciones de 2^{32} bytes.

RF 3.1 Memoria de código se debe poder cargar el código en la memoria, en el espacio del usuario, de manera que el registro EIP pueda apuntar a ella.

RF 3.2 Pila se debe simular la pila, pudiendo almacenar en ella valores y realizar las operaciones básicas (PUSH, POP).

RF 3.3 Segmentación se deben simular las estructuras necesarias para ofrecer una segmentación flat, es decir, segmentos existentes pero ignorables.

RF 4 Llamadas al sistema se deben poder ejecutar llamadas al sistema siempre y cuando el sistema operativo *host* sea Linux.

RF 4.1 Implementación se deben implementar los mecanismos necesarios para realizar la *trap* que desemboque en la ejecución de la llamada al sistema.

RF 4.2 Traducción se deben traducir los números de llamada al sistema entre 32 bits y 64 bits (X86 y X86_64).

RF 4.3 Argumentos se deben traducir los argumentos de las llamadas al sistema de 32 bits (4 bytes) a 64 bits (8 bytes), en especial los punteros a memoria simulada y los structs.

RF 5 Manejo de ficheros se deben poder manipular ficheros ejecutables, para poder cargarlos en memoria.

RF 5.1 ELF-32 se deben poder cargar ejecutables ELF-32 en memoria para su ejecución por el simulador.

RF 5.1.1 Loader se debe poder cargar el programa en memoria, obteniendo primero las cabeceras del fichero ELF para poder cargarlo a continuación.

RF 5.1.2 Nombres de funciones se deben poder extraer los nombres de las funciones del fichero ELF, así como sus direcciones para mostrarlas en cada instrucción. De esta manera cada instrucción mostrará en que función está, y cuantos bytes dentro de esa función.

RF 5.1.3 Obtener EIP se debe poder obtener el valor inicial para el registro EIP una vez se haya cargado el programa en memoria, para poder proceder a su ejecución.

RF 5.1.4 Preparar la pila se debe preparar la pila con los valores que incluiría el kernel real. Es decir, se deben cargar variables de entorno, valores auxiliares, y argumentos del programa, con una alineación correcta.

RF 5.2 Raw se deben poder cargar ejecutables raw (sin cabeceras) en memoria para su ejecución por el simulador.

RF 5.2.1 Loader se debe cargar la sección ejecutable en memoria para su posterior ejecución.

RF 5.2.2 Obtener EIP se debe obtener el valor inicial para el registro EIP (generalmente `0x00000000`) para que apunte a la primera instrucción del programa, para poder empezar la ejecución.

RF 6 Interfaz el simulador debe tener una interfaz gráfica que permita al usuario interactuar con ella y visualizar el estado actual del simulador.

RF 6.1 Ventanas la interfaz debe estar organizada en ventanas (recuadros), donde cada ventana muestre una información distinta.

RF 6.1.1 Registros se deben mostrar cada uno de los registros, junto a sus valores, en una de las ventanas de la interfaz.

RF 6.1.2 Código se deben mostrar la instrucción apuntada por el EIP, junto a las contiguas, en una de las ventanas de la interfaz.

RF 6.1.3 Pila se debe mostrar el valor en la cima de la pila, así como los valores siguientes, en una de las ventanas de la interfaz.

RF 6.1.4 Terminal se debe mantener una ventana como "terminal", para las opciones que requieran escribir.

RF 6.2 Comandos (opciones) se debe poder ejecutar una serie de opciones sobre el simulador, ya sea para controlar el flujo o para visualización del estado del simulador.

RF 6.2.1 Step se debe poder ejecutar una sola instrucción.

RF 6.2.2 Continue se debe poder ejecutar tantas instrucciones como sea necesario hasta alcanzar un breakpoint.

RF 6.2.3 No Enter se debe poder ejecutar tantas instrucciones como sea necesario hasta alcanzar la siguiente instrucción (evitar JMP's o CALL's).

RF 6.2.4 Breakpoint se debe poder añadir un nuevo breakpoint.

RF 6.2.5 Delete se debe poder eliminar un breakpoint existente.

RF 6.2.6 Find se debe poder buscar una dirección de memoria y mostrarla en la interfaz.

RF 6.2.7 Show Hex se debe poder mostrar el contenido hexadecimal de una dirección de memoria.

RF 6.2.8 Show String se debe poder mostrar la *string* apuntada por una dirección de memoria.

Requisitos No Funcionales

A continuación se muestra la especificación de Requisitos No Funcionales (RNF).

RNF 1 Separar ventanas de la interfaz se deben separar las ventanas de la interfaz con límites diferenciados, de manera que cada ventana quede contenida en una caja o recuadro.

RNF 2 Resaltar EIP se debe resaltar en la ventana de código la instrucción apuntada por el EIP actual, de manera que se sepa que va a ser la próxima en ejecutarse.

RNF 3 Resaltar registros modificados se deben resaltar en la ventana de registros, los registros que hayan sido modificados con respecto a la impresión anterior.

RNF 4 Colorear direcciones de memoria se deben colorear las direcciones de memoria de las ventanas de código y pila para diferenciarlas de su contenido.

RNF 5 Colorear funciones se deben colorear los nombres de las funciones (en otro color distinto que no se haya usado) para diferenciarlas de las instrucciones a la que identifican.

Apéndice C

Especificación de diseño

A continuación, especificaremos los aspectos más importantes del diseño del programa propuesto por este trabajo, que implementa un simulador funcional del microprocesador Intel 80386.

C.1. Introducción

El objetivo de este capítulo es determinar cómo ha sido construido el simulador. Desde como han sido implementadas las estructuras de datos simuladas, como el diseño arquitectónico del programa, pasando por todos los procesos y algoritmos que intervienen en la tarea de simular un programa compilado para el i386. En las siguientes secciones, explicaremos con todo lujo de detalles todos los pormenores del proceso de diseño.

C.2. Diseño de datos

Para el diseño de los datos, han sido utilizados los tipos de datos definidos por el fichero de cabecera `stdint.h`. Este archivo de cabecera define los tipos de datos sin signo `uint8_t`, `uint16_t`, `uint32_t` y `uint64_t`, así como sus variantes de tipo entero.

Estos tipos han sido utilizados en todo el programa, de manera que el tamaño de las variables sea estático, dependiente únicamente del número de bits que indique la variable, ya que de unas distribuciones de sistemas operativos a otros, puede variar el tamaño en bits de los tipos primitivos,

como `int`, `char`, que eran las alternativas a usar, con sus variantes `signed` e `unsigned`.

Registros

Los registros han sido implementados con la ayuda del fichero de cabecera `stdint.h`. Dichos registros son, excepto en el caso de los selectores de segmento que son de 16 bits, casi en su totalidad de 32 bits.

De igual manera, por simplicidad del código y de manipulación de los datos, todos los registros (incluidos los selectores de segmento que deberían ser de 16 bits) han sido implementados como variables globales de tipo `uint32_t`. Dichos registros son los siguientes.

Propósito del registro	Registro 32 bits	Registro 16 bits	Registros 8 bits
Acumulador	EAX	AX	AH y AL
Propósito general	EBX	BX	BH y BL
Propósito general	ECX	CX	CH y CL
Propósito general	EDX	DX	DH y DL
Base Frame Pointer	EBP	BP	-
Source Index	ESI	SI	-
Destination Index	EDI	DI	-
Stack pointer	ESP	SP	-
Instruction Pointer	EIP	IP	-
State Flags	EFLAGS	-	-
Selector de Segmento	-	CS	-
Selector de Segmento	-	DS	-
Selector de Segmento	-	ES	-
Selector de Segmento	-	GS	-
Selector de Segmento	-	FS	-
Selector de Segmento	-	SS	-

Tabla C.1: Registros del i386

La declaración de dichas variables asociadas a registros es algo parecido a lo siguiente.

```
uint32_t eax, edx, esp, esi, eip, ecx, ebx, ebp, edi, eflags;
uint32_t cs, ds, es, gs, fs, ss;
```

Tal como se ha mencionado antes, los selectores de segmento pueden ser definidos como `uint16_t`, aunque por simplicidad en el código y por estandarización, la manipulación de los registros está pensada para valores de 32 bits.

Como han sido implementados como variables globales, en principio solo serían visibles desde su propio fichero `.c`, sin embargo, hay una manera mediante la cual otros ficheros de código fuente puedan acceder a ellos, y es redefinirlos como variables globales externas (mediante la palabra reservada `extern`). Esto indica que son variables globales pero que están definidas y/o inicializadas en otro fichero de código fuente. La siguiente expresión sería un ejemplo de redefinición de una variable global para su uso en otro fichero.

```
extern uint32_t eax, edx, esp, esi, eip, ecx, ebx,
        ebp, edi, eflags, ss, es, gs, cs, ds, fs;
```

Todos los registros están declarados de manera principal en el fichero `instr.c`, menos el registro EFLAGS, que está declarado en el fichero `flags.c`.

De manera adicional, hay instrucciones específicas que pueden utilizar variantes de los registros de 32 bits. Por ejemplo, los 16 bits bajos del registro EAX corresponden al registro AX, y dicho registro puede dividirse en los registros AH (A High) y AL (A Low), que corresponden a sus 8 bits más significativos y a sus 8 bits menos significativos, respectivamente. Esta diferenciación se da en la mayoría de los registros de propósito general.

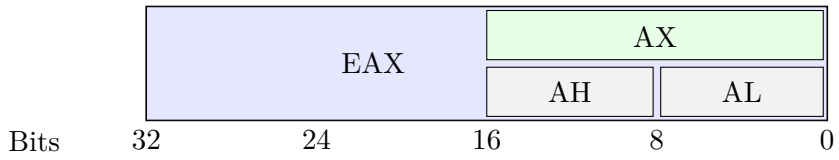

Figura C.1: Registros dentro de EAX

Podemos obtener punteros a estos registros de la siguiente manera.

```
uint8_t * al = (uint8_t *)&eax;
uint8_t * ah = ((uint8_t *)&eax)+1;
uint16_t * ax = (uint16_t *)&eax;
```

El puntero a AH se calcula igual que el puntero a AL pero sumándole 1 ya que el puntero es de `uint8_t`, por lo que si le sumamos uno, aumentará en 8 bits, dejando el puntero justo donde comienza AH.

Memoria

Como se ha mencionado en varias ocasiones a lo largo de este documento, la memoria del simulador propuesto por este trabajo se implementa como un conjunto de direcciones de memoria simuladas. Dicho espacio de direcciones es de 2^{32} bytes. La memoria del simulador se concibe como un puntero a una dirección de memoria de la máquina host, que tiene reservados 2^{32} bytes. Es decir,

```
uint8_t * mem;
...
mem = (uint8_t *)calloc(sizeof(uint8_t), UINT32_MAX );
```

De esta manera, la memoria se concibe como un array de 4.294.967.296 bytes [1], de manera que el puntero que se almacena corresponde a la dirección de memoria simulada `0x00000000`, y para acceder a una dirección de memoria cualquiera, por ejemplo, `0x12345678`, usaríamos la siguiente expresión.

```
uint8_t myByte = mem[0x12345678];
```

Esta expresión hace lo siguiente. Obtiene el valor que hemos indicado dentro de los corchetes, en este caso `0x12345678`, y lo multiplica por el tamaño del tipo del puntero en bytes, obtenible con la función `sizeof()`. En el caso de los bytes (el tipo `uint8_t`) es de 1. Con esto se obtiene el *offset*, que es el valor que debe ser sumado al puntero inicial para obtener el puntero al dato que queremos obtener. Por ello, la expresión anterior sería equivalente a la siguiente, siempre que el puntero `mem` sea de tipo `uint8_t *`.

```
uint8_t myByte = *(mem + 0x12345678);
```

El asterisco es un operador utilizado para desreferenciar un puntero, es decir, obtener el dato al que apunta el puntero. Primero se resuelve la expresión entre paréntesis y luego se realiza la operación de desreferenciación. En la primera expresión, a diferencia de esta segunda, la desreferenciación va implícita en el uso de los corchetes.

[1] Para reservar la memoria se utiliza la función `calloc()` en vez de `malloc()` para que al acceder a las direcciones de memoria, estén inicializadas a `0x00`, ya que con `malloc` estarían indefinidas y podrían ser interpretadas como código.

Para entender mejor lo que estamos haciendo, supongamos que nuestro puntero `uint8_t * mem` apunta a la dirección de memoria de 64 bits (de la máquina host) `0x56dba00000000`. Supongamos también que queremos acceder de nuevo a la dirección `0x12345678`.

- La dirección simulada `0x00000000` (la primera dirección de la memoria simulada) corresponde a la dirección `0x56dba00000000` real.
- La dirección simulada `0xFFFFFFFF` (la última dirección de la memoria simulada) corresponde a la dirección `0x56dbaffffffff` real.

Por lo tanto, para obtener el puntero al dato que queremos obtener, tendremos que sumar el puntero de la memoria (aquel que apunta a `0x00000000`), y sumarle la cantidad de bytes correspondientes. En el caso de que el tipo apuntado por el puntero sea de 1 byte (como `uint8_t`), el offset a sumar coincide con la dirección de memoria simulada.

```
0x56dba00000000 + 0x12345678 = 0x56dba12345678
```

La dirección de memoria que contiene el byte de la dirección simulada `0x12345678` sería la `0x56dba12345678`.

Sin embargo, en el contexto del simulador y su acceso a memoria, ocurre lo siguiente. El procesador puede leer y escribir de memoria bytes, palabras (*word*) o palabras dobles (*doubleword*).

- Los bytes ocupan 1 byte (8 bits).
- Las palabras ocupan 2 bytes (16 bits).
- Las palabras dobles ocupan 4 bytes (32 bits).

Debemos tener en cuenta que la memoria está implementada como un "array" de bytes. Por ello, utilizando lo mencionado anteriormente, si lo que queremos es leer o escribir una palabra o una palabra doble, deberemos convertir el puntero al tipo correspondiente antes de desreferenciarlo. Es decir, una vez obtenemos la dirección de memoria que contiene el dato que queremos, debemos convertir el puntero al tipo de puntero del dato que esperamos. Esto se debe a que es la manera que tiene el compilador de saber cuantos bytes debe leer del puntero. El orden de los bytes leídos (cuál va a ser el más significativo) depende de si usamos *Little Endian* o *Big Endian*.

Si vemos la Figura C.2, podemos entenderlo de la siguiente manera. Supongamos que las direcciones de memoria que aparecen en la figura son direcciones correspondientes a la memoria reservada para el puntero `mem`.

```
uint8_t myByte = *(mem + 0x0);
uint16_t myWord = *((uint16_t *)(mem + 0x8));
uint32_t myDWord = *((uint32_t *)(mem + 0x10));
```

Si imprimiéramos las variables, veríamos que tendrían los siguientes valores.

```
myByte  : 0x01
myWord  : 0x0302
myDWord : 0x0D0C0B0A
```

Esto se debe a que cuando el valor a leer es de más de un byte, el primer byte que se lee (y el que corresponde a la dirección de memoria que se especifica) se almacena en el byte menos significativo de la variable, mientras que el último byte leído se convierte en el byte más significativo de la variable.

Puede llegar a ser confuso ya que si vemos la memoria ordenada, el byte menos significativo está más a la izquierda (leemos de izquierda a derecha), pero en los valores que leemos, el byte menos significativa corresponde a las dos cifras de más a la derecha (por cómo interpretamos los números), siempre que usemos *Little Endian* (que es usado por el i386).

Address	**Content**							
0x00000000	0x01	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x00000008	0x02	0x03	0x00	0x00	0x00	0x00	0x00	0x00
0x00000010	0x0A	0x0B	0x0C	0x0D	0x00	0x00	0x00	0x00

Figura C.2: Ejemplo de memoria para lectura según el tipo de puntero.

Por último, al igual que los registros, la memoria está definida como una variable global definida en el fichero `instr.c`, y puede ser usada desde otros ficheros de código fuente utilizando la siguiente redeclaración.

```
extern uint8_t * mem;
```

Pila (*Stack*)

La pila (o *stack*) del simulador se implementa como una zona reservada de memoria dentro del espacio de memoria del *Kernel*. El espacio del *Kernel* va desde la dirección `0xC0000000` a la dirección `0xFFFFFFFF`, tal como se puede ver en la Figura C.3.

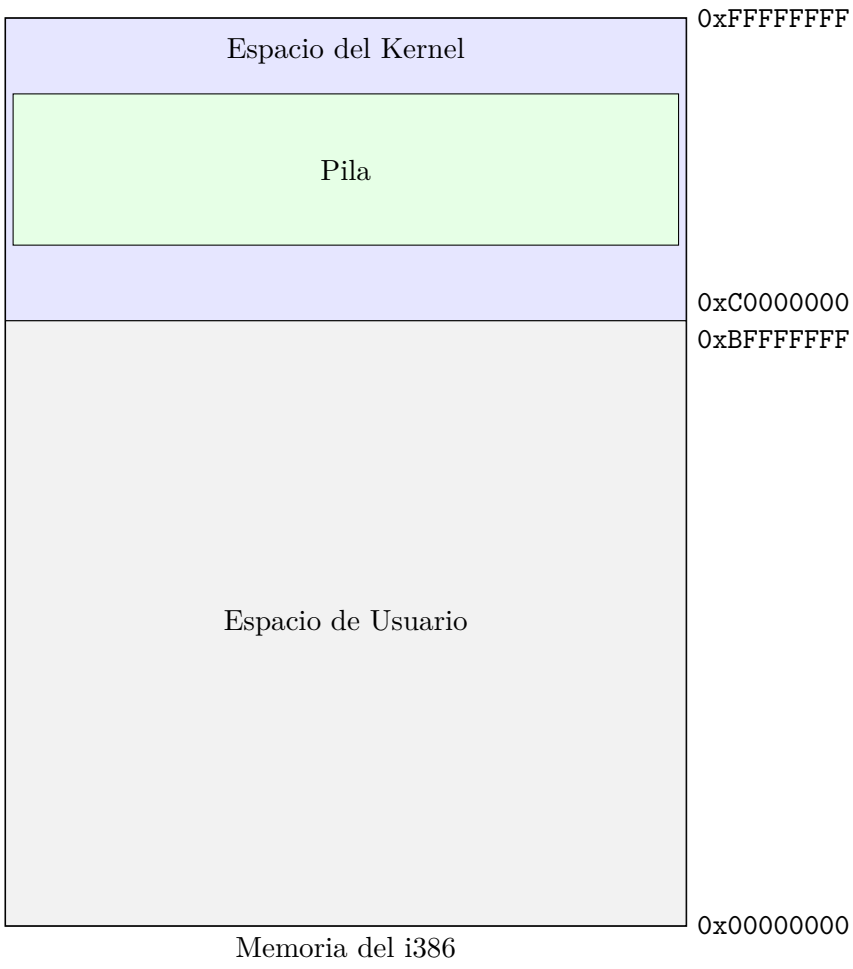

Figura C.3: Esquema de memoria

En el código, esto se define mediante dos variables globales en `instr.c` que almacenan la dirección de inicio y final de la pila. La cima de la pila (*STACK_BOTTOM*) es el valor inicial de la pila y el más alto de los dos,

mientras que el suelo (*STACK_TOP*) [2] es el tope hasta el que puede crecer la pila, y el menor de los dos valores. Se ha elegido un tamaño fijo de la pila de `0x22000` bytes, es decir, 139.264 bytes.

```
STACK_BOTTOM = 0xFFFFA000;
STACK_TOP = STACK_BOTTOM - 0x22000;
esp = STACK_BOTTOM;
```

Se pueden realizar dos operaciones fundamentales sobre la pila. Lo normal es que los tamaños de los valores de dichas operaciones sean múltiplos de 4 bytes, aunque pueden haber excepciones. En ocasiones (como al inicio de funciones) se suele "alinear la pila", que consiste en dejarla en un valor múltiplo de 4.

PUSH Almacena un valor en la pila. Corresponde a restar un número de bytes del registro ESP (para hacer espacio para el valor), y escribir dicho valor.

POP Recupera un valor de la pila. Corresponde a obtener un valor y luego sumar al registro ESP el tamaño en bytes de ese valor.

Por ejemplo, si quisiéramos almacenar un valor de 4 bytes en pila, realizar alguna operación, y luego sacar ese valor de la pila, haríamos lo siguiente. Las funciones `read32()` y `write32()` son funciones definidas en `instr.h` e implementadas en `instr.c` que leen o escriben un valor de 4 bytes (32 bits) en una determinada dirección de memoria en *Little Endian*.

```
esp -= 4;
write32(esp, valor);
...
valor = read32(esp);
esp+=4;
```

Al restar o sumar valores al registro ESP (*Stack Pointer*), se debe comprobar antes de la operación que el nuevo ESP que obtendríamos no residiera fuera

[2] Los sobrenombres "BOTTOM" y "TOP" pueden ser confusos ya que "BOTTOM" es el valor más alto y viceversa, pero debe plantearse la pila como un objeto que crece hacia abajo. Por ejemplo, podríamos concebir la pila como una estantería que rellenamos desde la balda de arriba hacia la de abajo, por lo que la base es la balda de arriba, y el tope es la balda que esté más abajo, ya que una vez la hayamos rellenado, no podremos añadir nada más.

de los límites establecidos de la pila. Si así fuera, podríamos leer o escribir el valor, pero sin llegar restar ni sumar nada, de manera que se modifique la última dirección de pila posible, pero no ajuste el *Stack Pointer*.

```
esp -= (esp - 4 < STACK_TOP)? 0 : 4;
write32(esp, valor);
...
valor = read32(esp);
esp += (esp + 4 > STACK_BOTTOM)? 0 : 4;
```

De esta manera podría seguir usándose la pila, aunque sobreescribiendo los valores límite, pero sin que hubiera un *overflow* de memoria.

GDT

La GDT o *Global Descriptor Table* es una estructura de datos que almacena una serie de entradas relacionadas con la segmentación, llamadas descriptores. Cuando un programa necesita resolver una dirección de memoria usando segmentación, usa un registro de selector de segmento (CS, ES, DS, FS, GS o SS). Dichos selectores son registros de 16 bits que referencian a un descriptor de la GDT.

Cuando intentamos obtener una dirección de memoria efectiva, lo hacemos usando un *offset* y un selector de segmento. El selector de segmento apunta al descriptor. Cada entrada o descriptor de la GDT tiene una información específica, aunque la información más relevante la tienen los siguientes campos.

Base dirección base del segmento.

Límite dirección límite del segmento. La suma de la base del segmento y el *offset* no debe superar este límite.

Privilegios privilegios necesarios para acceder a la memoria comprendida en el segmento.

Sin embargo, como el simulador ha sido concebido con arquitectura FLAT, los selectores y descriptores de segmento, aunque están implementados tienen como base `0x00000000` y límite de `0xFFFFFFFF`, por lo que es como si no estuvieran. Igualmente, los privilegios son ignorados al implementar la arquitectura FLAT.

Para definir la GDT, hemos definido los siguientes macros y tipos.

```
#define GDT_ENTRIES 0x9
#define GDT_ADDR 0xC0FFF000

typedef struct{
    uint16_t limit_low;
    uint16_t base_low;
    uint8_t base_mid;
    uint8_t access;
    uint8_t granularity;
    uint8_t base_high;
    uint8_t used;
}GDT_Descriptor;

typedef struct{
    uint16_t limit;
    uint32_t base;
}GDTR;
```

El tipo `GDT_Descriptor` corresponde a cada uno de los descriptores que van a componer la GDT, mientras que el tipo `GDTR` corresponde al registro GDTR (*Global Descriptor Table Register*), que almacena la dirección de memoria donde está alojada la GDT, y dicha dirección de memoria está definida por el macro `GDT_ADDR`, por lo que durante la inicialización, asignaremos el valor de `GDT_ADDR` al registro GDTR.

Durante la inicialización, se ejecuta la siguiente secuencia de líneas.

```
GDT_Descriptor * gdt = (GDT_Descriptor *)(mem + GDT_ADDR);
init_gdt(gdt);
gdtr.base = GDT_ADDR;
gdtr.limit = GDT_ENTRIES * sizeof(GDT_Descriptor);
```

Con ello, estamos asignando al puntero `GDT_Descriptor * gdt` la dirección de memoria de la máquina host donde va a residir la GDT. A continuación, usamos dicha dirección de memoria en la función `init_gdt()`, que inicializa los descriptores, y a continuación inicializamos el registro GDTR con la dirección de inicio de la GDT, y la dirección de fin, que corresponde al inicio sumado al número de descriptores por su tamaño.

Tabla de *Syscalls*

En cuanto a la versión de Linux, existe una estructura de datos que funciona como tabla de llamadas al sistema, de manera que mediante el número de la llamada al sistema, se puede llamar directamente a la función que la implementa y ejecutarla. Para ello, primero es necesario definir los tipos que vamos a utilizar.

Se ha definido el tipo `Syscall32bits`, que define la signatura que van a llevar las funciones que implementan las *syscalls* de 32 bits. Toman 7 argumentos, el primero corresponde al registro EAX, que contiene el número de syscall, y el resto corresponden a los registros que contiene los argumentos (EBX, ECX, EDX, ESI, EDI, EBP). Dicho tipo viene definido en `trad_syscall.h`.

Esta signatura en específico no se ha elegido durante el desarrollo de este proyecto sino que viene dada por la ABI del i386, donde tanto el número de la llamada al sistema como el resultado de la misma se obtienen del registro EAX, mientras que los argumentos se toman tal como se ha mencionado anteriormente.

```
typedef uint32_t(*Syscall32bits)(uint32_t *nr, uint32_t *arg1,
    uint32_t *arg2, uint32_t *arg3, uint32_t *arg4,
    uint32_t *arg5, uint32_t *arg6);
```

Tanto el número de llamada al sistema `uint32_t * nr` (correspondiente al registro EAX), como cada uno de los 6 argumentos de la llamada han sido pasados como punteros, para que puedan ser modificados durante la *syscall* si es necesario. Por otro lado, en el fichero `trad_syscall.c`, está definida la tabla que contiene los punteros a todas las funciones que implementan las *syscalls*.

```
Syscall32bits syscalls[]= {
    do_restart_syscall, do_exit, do_fork, do_read, do_write,
    do_open, do_close, unimpl, do_creat, do_link, do_unlink,
    do_execve, do_chdir, do_time, do_mknod, do_chmod, do_lchown,
    ...
    unimpl
};
```

La implementación de llamadas al sistema ha sido parcial, definiendo únicamente aquellas que sean necesarias para ejecutar un programa sencillo, como

puede ser uno que lea o escriba por terminal. Los índices de las funciones en la tabla corresponden con su número de *syscall* asociado.

La referencia a algunas llamadas al sistema han quedado con la función `unimpl()` no porque no hayan sido implementadas, sino porque han desaparecido de 32 bits a 64 bits. Es decir, ya no existen en 64 bits, por lo que no podrán ser ejecutadas en la máquina host.

Tabla de Instrucciones

Para poder elegir de manera automática la instrucción a ejecutar, se ha hecho lo siguiente. Una vez decodificada la instrucción pertinente, se obtiene una cadena (puntero `char *`) que contiene el nemotécnico de la instrucción. Los nemotécnicos son únicos así que no hay confusión posible. Una vez con dicha cadena, se compara uno a uno con todos los nemotécnicos de todas las instrucciones, hasta encontrar una coincidencia.

Existe un array de nemotécnicos, y un array de punteros a funciones, cuyos índices coinciden, de manera que la cadena `"add"` del array de nemotécnicos y el puntero a función `add_i` tienen el mismo índice en sus respectivos arrays.

```
const char *inss[] = {
"aaa","aad","aam","aas","adc","add","and", "arpl",
...
"rep outs", "rep stosb","rep stosw","rep stosd",""};

Instruction instructions[] = {
aaa_i, aad_i, aam_i, aas_i, adc_i, add_i, and_i, arpl_i,
...
rep_outs_i,rep_stos_i, rep_stos_i, rep_stos_i, NULL};
```

De esta manera, podemos ejecutar una instrucción a partir de su nemotécnico, tal como se ve en la Figura C.4.

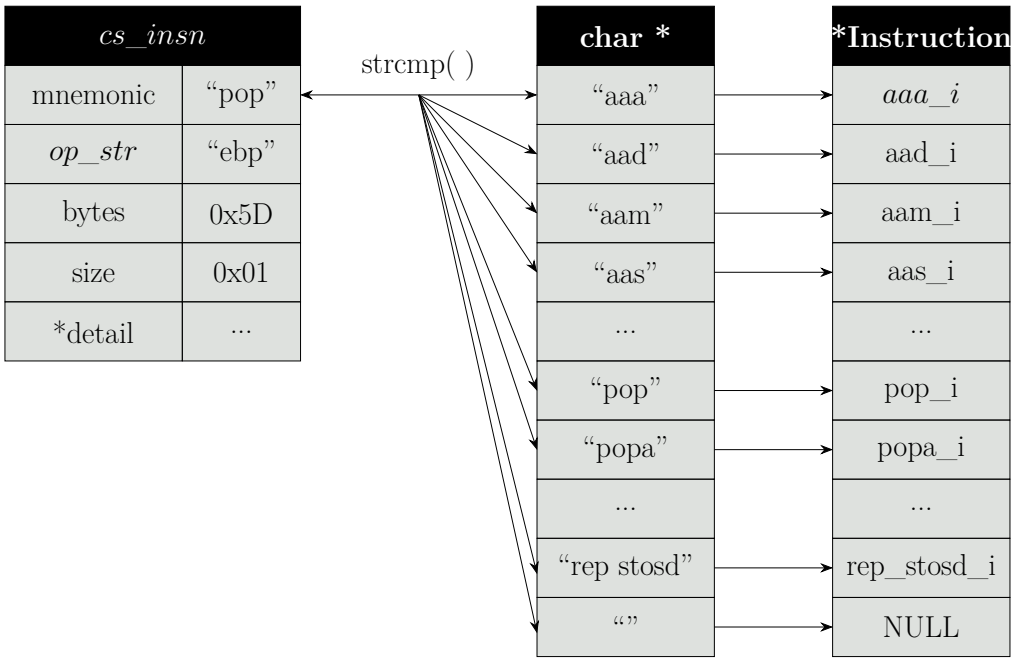

Figura C.4: Funcionamiento del dispatcher de instrucciones

El tipo `Instruction` es un tipo personalizado que define un estándar de signatura para todas las instrucciones. Todas las funciones que implementan instrucciones tienen como nombre su nemotécnico más un sufijo `_i`, todas toman como argumento una instrucción del tipo `cs_insn`, y todas devuelven un `int`. Dicho tipo está definido en `instr.c`.

```
typedef int(*Instruction)(cs_insn *insn);
```

Con todo esto, funciona la función `dispatcher`, que toma como argumentos una instrucción `cs_insn` y un puntero a cadena (`char *`), con ello busca una coincidencia en el array de cadenas `inss` (Instruction Strings), y si la encuentra, usa ese mismo índice para llamar a la función que esté en esa misma posición en el array de instrucciones `instructions`. Esta función propaga el valor devuelto por la instrucción que acaba de ejecutar, por si fuera un código especial que indica que el programa ha acabado.

```
int dispatcher(char * mnemonic, cs_insn * insn){
    for (int i = 0; i< NINS ; i++){
        if(strcmp(inss[i], mnemonic) == 0){
            return instructions[i](insn);
```

```
            }
        }
        return -1;
    }
```

`NINS` es el macro que define el tamaño de ambos arrays, tanto el de cadenas como el de punteros a funciones. Está definido en `instr.h`.

```
#define NINS 205
```

Si quisiéramos implementar la técnica de mitigación de NX (*Non-Executable*) [2], que impide ejecutar instrucciones que residan en la pila, debería implementarse en esta misma función, con una comprobación parecida a la siguiente justo antes del bucle.

```
/* Non-Executable Mitigation Technique */
if(eip >= STACK_BOTTOM && eip <= STACK_TOP){
    /* Instruction resides on the stack*/
    return 0xdeadbeef;
}
```

Donde `0xdeadbeef` es un código especial que indica que la ejecución del programa debe terminar.

C.3. Diseño arquitectónico

La arquitectura se ha planteado como un programa de terminal (en la rama de Linux) que funcione como simulador del Intel 80386, pero que a su vez, con unas ligeras modificaciones, pueda ser utilizado como una librería para otros programas (en la rama de Windows). La estructura global del sistema se presenta como un conjunto de ficheros fuente C modular donde cada uno tiene su responsabilidad propia, y es usado por sí mismo o por otros ficheros con un mayor nivel de responsabilidad. Cada módulo está conformado por un fichero de cabecera (`.h`) y su implementación en código fuente (`.c`). Ambos ficheros comparten nombre pero difieren en la terminación.

Cada módulo por separado implementa mecanismos asociados a su responsabilidad pero que son usados por él mismo o por los ficheros que tiene por encima en el grafo de dependencias. De igual manera, el módulo

principal (ya sea `proc` o `main`) es el que se encarga de coordinar al resto de módulos.

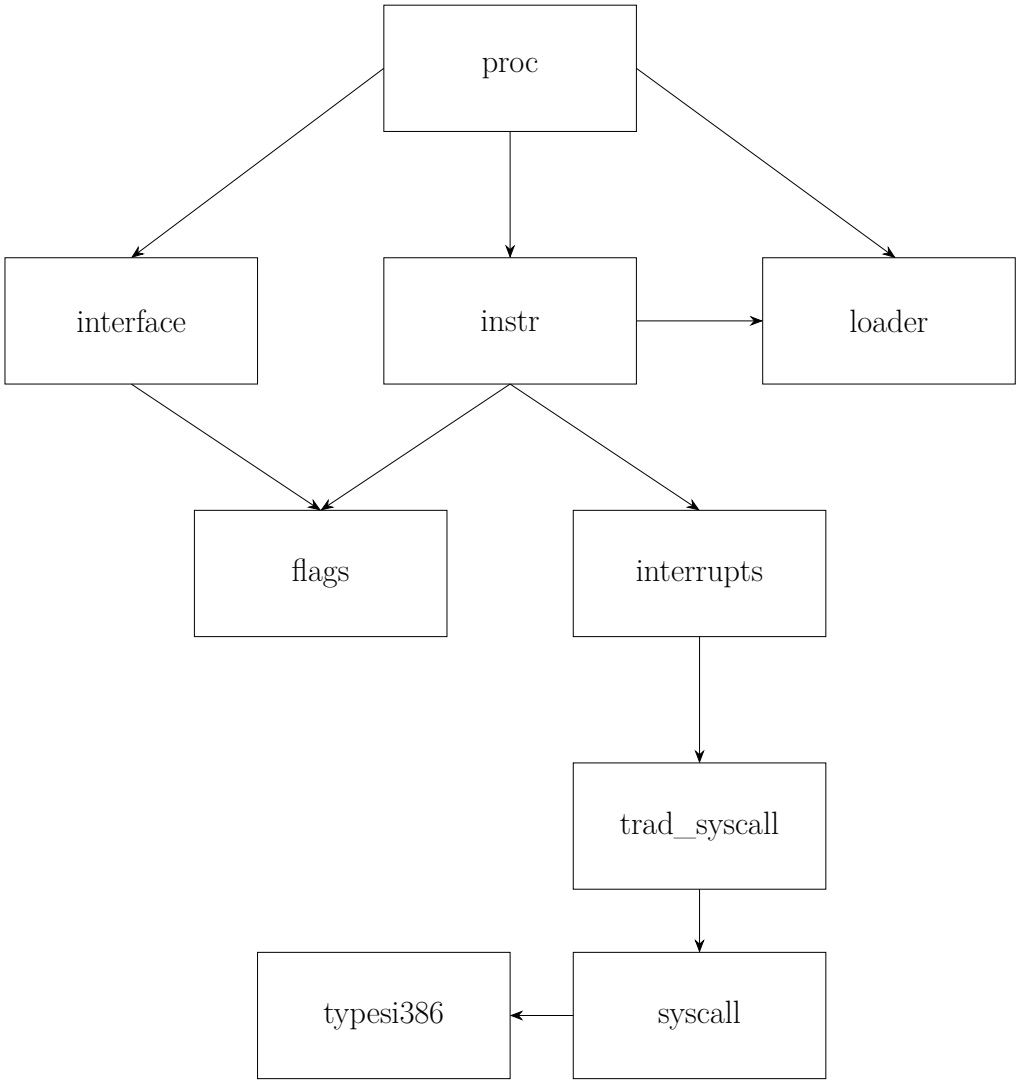

Figura C.5: Estructura de módulos

Esta arquitectura basada en módulos y responsabilidades ha sido inspirada por los patrones de desarrollo de Java y la arquitectura MVC (Modelo Vista Controlador) [3], donde existe un estado interno del modelo (registros, memoria), que se le ofrece al usuario a través de vistas, y este interactúa por medio del controlador. Sin embargo, no ha podido implementarse de manera pura debido a las siguientes limitaciones.

- El lenguaje C no permite la creación de clases ni un encapsulamiento tan fuerte como sí permite Java u otros lenguajes orientados a objetos, por lo que las diferencias entre Vista y Controlador pueden ser difusas.
- A diferencia del MVC clásico, la interacción del usuario está limitada a una serie de operaciones (Step, Breakpoint,...).
- Los patrones de diseño no son tan claros.

Estas limitaciones son la consecuencia de la elección de C como el lenguaje de desarrollo de este proyecto, que son el precio a pagar por estar más próximos al Kernel del sistema operativo y del bajo nivel, algo muy importante para el proyecto. De igual manera, el código presenta un acoplamiento entre módulos más alto de lo deseado.

En la rama de Windows del proyecto, empieza a ser más visible la inspiración en MVC, donde el modelo es la librería DLL con su estado interno, la vista es la ventana que se le muestra al usuario, y el controlador sería el fichero que funciona como interfaz entre ambos y que se encarga de traducir las llamadas de funciones entre el DLL y el programa `.NET`. De igual manera, para el DLL de Windows, ha sido eliminado el módulo `trad_syscall` y todos los que cuelgan de él.

Módulos y responsabilidades

Los siguientes módulos son comunes tanto a la rama de Linux como a la rama de Windows. Dichas dependencias funcionan exactamente igual sin importar la plataforma.

- **instr** define los registros, las estructuras de datos no específicas (memoria, pila, gdt) y se encarga de la ejecución de las instrucciones. Expone las funciones `dispatcher()` e `initialize()`, que ejecutan una instrucción e inicializa la memoria, respectivamente. Su responsabilidad reside en el manejo de los registros, la memoria (incluyendo la pila), y la ejecución de las instrucciones.
- **flags** define el registro de banderas (EFLAGS) y le otorga el valor por defecto. Expone las funciones que permiten modificar dicho registro (`setFlag()`, `clearFlag()`, `testFlag()`), así como las funciones que determinan si una bandera debe activarse o no en función de los operandos y el resultado (`overflow()`, `adjust()`, ...). Su responsabilidad reside en el manejo de las banderas.

- **loader** se encarga de cargar el programa ejecutable en memoria, almacenar la información de las secciones ejecutables y preparar la pila para la ejecución del programa. Su responsabilidad reside en hacer de *loader* del programa, como haría el Kernel del sistema operativo. Expone las funciones `read_raw_file` y `read_elf_file`. En la versión de Windows, no se utiliza la última de las dos.

Por otro lado, los módulos específicos de Linux son los siguientes.

- **proc** es el fichero principal del programa. Se encarga de recoger las opciones del usuario y su entrada opción a opción. También define variables auxiliares y sobre todo, coordina el resto de módulos de cara a la ejecución completa del programa. Su responsabilidad es actuar como intermediario entre el usuario y el *core* del simulador, coordinando las interacciones entre ambos, y entre los propios componentes *core*.

- **interrupts** se encarga de la gestión de las interrupciones. Como las interrupciones al uso no han sido implementadas, es una especie de interfaz para la gestión de las interrupciones, por si se implementan más adelante. Sin embargo, sí que actúa como intermediario entre los módulos `instr` y `trad_syscall`, ya que las llamadas al sistema también se llaman mediante interrupciones. Expone la función `int_dispatcher()`.

- **syscall** se encarga de la ejecución de las llamadas al sistema, donde recibe unos argumentos correspondientes a los registros de 32 bits, y los traduce a valores de 64 bits para mandárselos a una llamada al sistema nativa de 64 bits. Expone todas las funciones de llamadas al sistema.

- **trad_syscall** se encarga de la traducción de los números de llamada al sistema en 32 bits a 64 bits. Su responsabilidad es dicha traducción, recibiendo un número de syscall que debe ejecutar. Expone un array de punteros a funciones del módulo `syscall`.

- **typesi386** se encarga de la traducción de tipos de 32 bits a 64 bits. Su responsabilidad es ayudar al módulo `syscall` en la traducción de argumentos de las llamadas al sistema.

Por último, los módulos específicos de Windows son los siguientes.

- **main** es el fichero principal de esta rama. Se encarga de coordinar al resto de módulos y sobre todo, exponer las funciones correspondientes a los casos de uso mediante un macro, para que cuando se compile el código como DLL, dichas funciones puedan ser llamadas desde fuera para ser ejecutadas por el DLL.
- **interrupts** es una variante de las interrupciones de Linux, donde las llamadas al sistema están recortadas, y su llamada devuelve un código de error que finaliza el programa.

Propagación de errores

A diferencia de otros lenguajes orientados a objetos como Java (o Python incluso), el lenguaje de programación empleado, C, no dispone de mecanismos de generación y manejo de excepciones. Por ello, se debe ser especialmente cuidadoso con el manejo de errores durante la ejecución del programa, ya que un acceso inválido a memoria provocaría un fallo de segmentación (*Segmentation Fault*) y el programa finalizaría de manera forzosa por parte del sistema operativo.

Por ello, se han realizado gran cantidad de comprobaciones a lo largo del código, para evitar estos errores. De igual manera, cuando sí que se detecta un error, se maneja de manera que la función en la que se encuentre, devuelve un código de error, `0xdeadbeef` en Linux y `0xf4` en Windows, y dicho valor retornado se va propagando de vuelta hasta alcanzar el módulo principal, que lo evalúa y si lo detecta, muestra un mensaje de error y finaliza el programa.

Pongamos el caso de la técnica de mitigación NX en la rama de Linux. Cuando el usuario activa dicha opción, y se comprueba que la instrucción a ejecutar reside en la pila, simulamos el mensaje de error que mostraría el sistema operativo si lo hiciéramos en un programa real, ejecutado sobre el sistema operativo real, y devolvemos el código de error.

```
if (nx){
    /* Instruction resides on kernel space */
    if(eip > STACK_TOP){
        /* Stack should not be executable */
        printf("Segmentation fault (core dumped)\n");
        return (uint32_t)0xdeadbeef;
    }
}
```

Una vez la función ha devuelto dicho valor, se comprueba desde el módulo principal para saltar a la finalización del programa.

```
res = dispatcher(ins[0].mnemonic, &ins[0]);

if (res == 0xdeadbeef){
     goto exit;
}

...
exit:
if (debugger){
  fclose(debugger);
}
free(mem);
return 0;
```

De esta forma se manejan los errores de manera general durante la ejecución del código, aunque los errores que pueden surgir son limitados y están muy controlados.

Interfaces (.h)

Los ficheros de cabeceras con extensión `.h`, pueden ser vistos como interfaces en el contexto de la programación en C. Su utilidad reside en declarar funciones, macros, tipos o constantes globales, que luego son implementados en otros ficheros, generalmente ficheros con extensión `.c`. Permiten organizar el código y generar una estructura modular basada el librerías o distintos ficheros que utilizan funcionalidades de otros.

En el presente proyecto, los ficheros de cabeceras han sido utilizados para definir tipos, macros, constantes, y sobre todo las signaturas de las funciones que va a exponer cada módulo. Por ejemplo, si queremos que la función `dispatcher()` del módulo `instr` pueda ser utilizado por el módulo principal, deberemos exponerla, es decir, incluir la signatura de dicha función en el fichero de cabecera `instr.h`, y posteriormente implementarla en el fichero de código fuente `instr.c`. En el fichero cabecera veríamos lo siguiente.

```
#include <stdint.h>
#include "flags.h"
```

```
#include <capstone/capstone.h>
#include <capstone/x86.h>

#ifndef INSTR_H
#define INSTR_H

int dispatcher(char * mnemonic, cs_insn * insn);

#endif
```

El macro `INSTR_H` sirve para ser definido cuando se realice la importación del fichero, por lo que si luego otro fichero también quiere importarlo, el macro ya estará definido, y por tanto el fichero importado. Con esto evitamos importar una misma cabecera varias veces.

Posteriormente, desde el fichero donde queramos importarlo, solo deberemos utilizar la directiva al preprocesador `#include`, con el nombre y ruta relativa al fichero de cabecera entre comillas dobles. Por ejemplo, desde el fichero `proc.c`.

```
#include "../lib/instr.h"
```

Las dependencias entre módulos se resuelven en tiempo de compilación. Por simplicidad y organización del código, todas las cabeceras han sido incluidas en un directorio llamado `lib`, mientras que todos los ficheros de código fuente han sido incluidos en un directorio llamado `src`, estando ambos directorios al mismo nivel en el respositorio.

Decisiones de diseño

El proceso de diseño ha sido especialmente complejo debido a la naturaleza del lenguaje C, y sus diferencias con otros lenguajes orientados a objetos donde aplicar patrones de diseño es más intuitivo, y más sencillo. C no dispone ni de clases, ni encapsulamiento fuerte, ni herencia, ni excepciones. De igual manera, se han tenido en cuenta algunos principios de diseño a la hora de realizar el diseño. En las decisiones de diseño también han influido los conocimientos adquiridos de manera teórica durante el Grado, así como los adquiridos de manera práctica durante experiencias profesionales y proyectos propios. Dichos conocimientos adquiridos de manera práctica han sido de especial importancia ya que han permitido un desarrollo basado en buenas prácticas que favorecen la productividad y evitan problemas futuros.

- **Responsabilidad única** : cada módulo está asignado a una responsabilidad única. La idea es que cada módulo se encargue de lo suyo y que lo haga bien, nada más. De esta manera se evitan cantidades monstruosas de código muy fuertemente acopladas, que son extremadamente complejas de reescribir.
- **Uso de tipos de tamaño invariable** : los tipos utilizados, tanto simples como estructurados, están basados en tipos de la librería `stdint.h`, cuyo tamaño en bits es invariable, de manera que no sean modificados en función de la librería usada para la compilación, como puede pasar con los tipos primitivos.
- **Jerarquía de dependencias** : cada módulo usa (o no) las funciones expuestas por otros módulos por debajo de él en el grafo de responsabilidad.
- **Diseño basado en interfaces** : los archivos de cabecera actúan como contratos que definen qué funcionalidades ofrece cada módulo, permitiendo cambiar implementaciones sin afectar al resto del sistema.
- **Minimización del acoplamiento** : se ha evitado que un módulo acceda a las estructuras de datos de otro donde no sea estrictamente necesario.
- **Gestión explícita de memoria** : reserva y liberación explícita de memoria cuando su uso esté finalizado, ya que C no dispone de un recolector de basura como sí tiene Java.
- **Control de errores explícito** : ante la ausencia de excepciones, las funciones devuelven códigos de error, obligando a su manejo a los módulos con más responsabilidad.
- **Protección frente a inclusiones múltiples** : todos los archivos de cabecera incluyen guardas (`#ifndef, #define, #endif`) para evitar redefiniciones y errores de compilación.

En adición a lo anteriormente mencionado, también se han incluido algunos patrones de diseño, de alguna manera u otra.

- **Patrón Strategy (Estrategia)** : se ha simulado mediante el uso de punteros a función, permitiendo seleccionar en tiempo de ejecución diferentes comportamientos sin modificar el código cliente. Podemos verlo en los *handlers* y *dispatchers* de las instrucciones o las llamadas al sistema.

- **Patrón Adapter (Adaptador)** : se han creado funciones intermedias para adaptar interfaces incompatibles entre módulos, facilitando la reutilización de código existente. Por ejemplo, encapsulando los *arrays* de punteros a funciones en una función adaptador.

- **Patrón Modular / Component-Based** : el sistema se ha dividido en módulos independientes con interfaces bien definidas, lo que facilita la reutilización y el testeo individual.

C.4. Diseño procedimental

El diseño del simulador se ha basado en diferentes módulos, cada uno con una responsabilidad única, y que exponen funcionalidades para que sean llamadas por otros módulos por encima en el grafo de responsabilidad. Esto les permite realizar las diferentes tareas necesarias para llevar a cabo la simulación de un programa.

A continuación, vamos a explicar como funciona internamente cada módulo, tanto su responsabilidad como desde donde son llamados. Cada módulo está compuesto por un fichero de código fuente con extensión `.c` y por su fichero cabecera con extensión `.h` correspondiente. Los ficheros fuente residen en la carpeta `src` y están emparejados cada uno con un fichero cabecera en la carpeta `lib`.

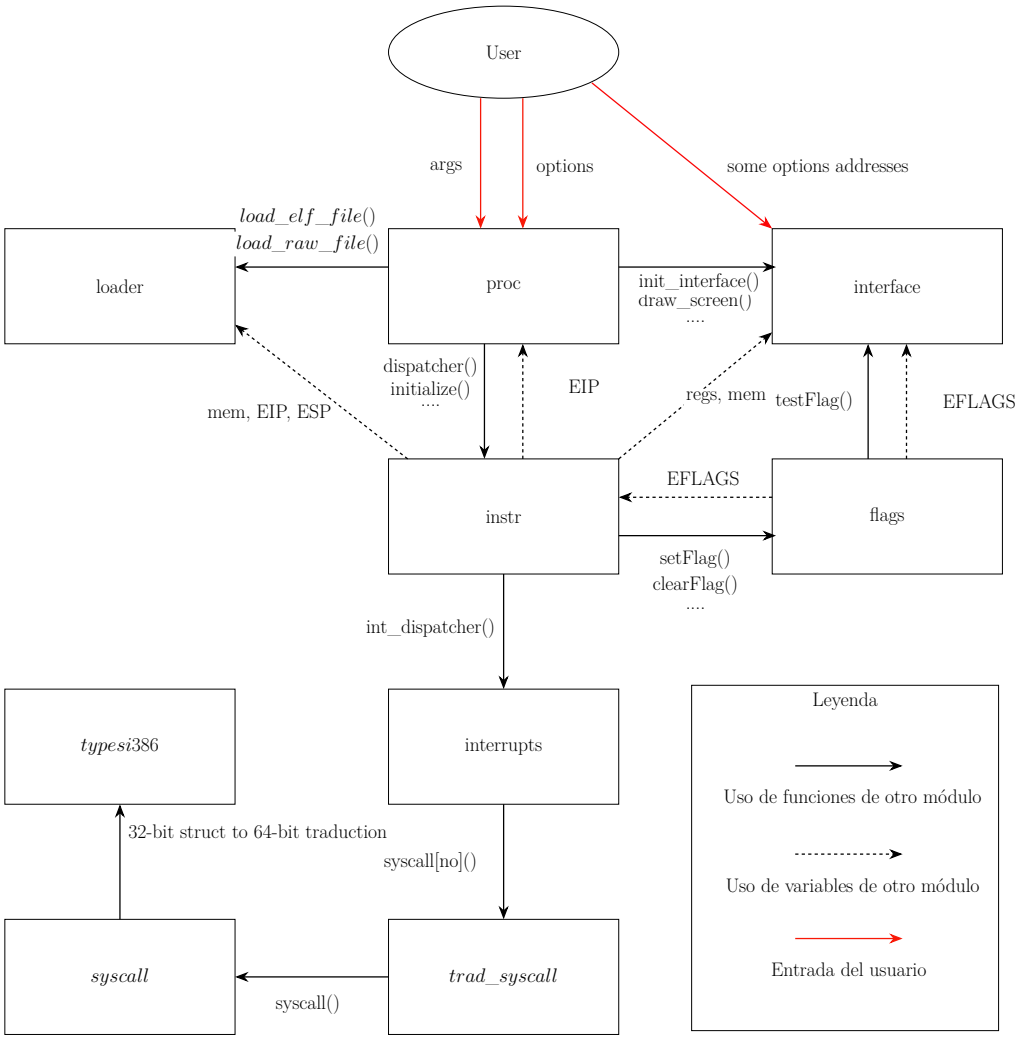

Figura C.6: Diagrama de interacción entre módulos

Tal como podemos ver en la Figura C.4, los módulos interactúan de la siguiente manera.

- **flags** define el registro EFLAGS y expone funciones que son llamadas para cambiar el valor del registro, comprobar su estado, o comprobar si el estado debe ser cambiado. El módulo `instr` emplea dichas funciones, y el valor del registro, ya que es necesario para la ejecución de algunas instrucciones, y otras modifican dicho registro. El módulo `interface` solo emplea el valor del registro y la función `testFlag()`,

para comprobar si cada una de las banderas está activa o no, y poder mostrarlo en la interfaz.

- **typesi386** define y expone funciones de traducción de tipos estructurados de 32 bits a 64 bits. Es usado por el módulo `syscall` para la traducción de argumentos de llamadas al sistema.
- **syscall** define y expone funciones que implementan llamadas al sistema, mediante la traducción de argumentos de 32 bits a 64 bits para ejecutar una llamada al sistema nativa. Es usado por el módulo `trad_syscall`, que recopila todas las funciones expuestas por este módulo en un vector.
- **trad_syscall** define y expone un vector con las llamadas al sistema recopiladas `syscalls[]`. Es usado por el módulo `interrupts` para la ejecución de llamadas al sistema cuando recibe una interrupción `0x80`.
- **interrupts** define y expone un *handler* de interrupciones llamado `int_dispatcher()`, donde recibe un número y ejecuta la función en esa posición de la IDT. Es usado por el módulo `instr` para la ejecución de interrupciones cuando recibe una instrucción `INT`.
- **instr** define los registros, la memoria, y expone las funciones `initialize()` para la inicialización, y `dispatcher()` para la ejecución de instrucciones. Es usado por el módulo `loader`, que emplea la memoria para cargar ahí el programa, utiliza el registro ESP para preparar la pila, e inicializa el registro EIP. El módulo `interface` utiliza el valor de sus registros para mostrarlos en la interfaz, así como las direcciones de memoria de la pila, determinadas por el registro ESP, y que también residen en memoria. El módulo `proc` utiliza las funciones expuestas, para la inicialización del simulador y la ejecución de funciones.
- **loader** define y expone las funciones que se encargan de cargar los ficheros ejecutables en memoria, preparar la pila, y almacenar la información de las secciones ejecutables. Es usado por el módulo `proc` para cargar el fichero en memoria y obtener las secciones ejecutables para poder desensamblarlas y obtener las instrucciones.
- **interface** define y expone las funciones relativas a la interfaz. Permiten imprimir la pantalla completa, poner la terminal en modo canónico o no, limpiar la pantalla, o solicitar datos al usuario de manera amigable a la interfaz, sin descuadrarla. Es usado por el módulo `proc` para imprimir el estado del programa simulado, los cambios en los registros, instrucciones, estado de la pila, etc.

- **proc** es el módulo principal. Obtiene los parámetros del usuario y coordina al resto de módulos para llevar a cabo la simulación del programa en función de lo que haya requerido el usuario. Actúa como intermediario entre el usuario y la lógica de simulación.

En el caso de los módulos que varían en la rama de Windows, interactúan de manera similar.

- **interrupts** define y expone la función `int_dispatcher()`, que al ser llamada devuelve siempre el valor especial `0xf4`, que es propagado hasta el programa que emplee el DLL e indica la finalización del programa. Sigue siendo usado por el módulo `instr`.
- **main** define y expone una serie de funciones asociadas a los casos de uso, para que puedan ser usadas de forma externa desde el programa que emplee el DLL. Llevan asociadas un macro que permiten su exposición al compilar dicho DLL.

Apéndice D

Documentación técnica de programación

D.1. Introducción

El presente proyecto es ha sido desarrollado con una filosofía *Open Source*, para ser utilizado de manera libre y gratuita por docentes y estudiantes de cualquier universidad. Por ello, el software y su documentación debe estar prevenidos para que otros programadores puedan entender el código, y llegar a modificarlo si fuera preciso. Esto implica la concepción de un manual de programador completo y detallado que permita hacer modificaciones personalizadas o mejoras significativas al código.

A continuación, se van a detallar los pormenores necesarios para la comprensión y posterior modificación del código. Como ya se ha mencionado anteriormente, existen dos versiones del simulador, una orientada a la ejecución en sistemas basados en Linux, y otra versión para Windows.

Cabe resaltar que ambas ramas comparten ciertos ficheros (a veces con ligeras modificaciones) que implementan funciones comunes a ambos entornos. Únicamente difieren aquellos ficheros que implementan funciones específicas de la arquitectura, como las llamadas al sistema, o la interfaz, ya que el Linux se plantea como una TUI (*Text-based User Interface*) incluida en el código C, mientras que en Windows se realiza una GUI (*Graphic User Interface*) por separado mediante .NET y Windows Forms. Véase la Figura D.1.

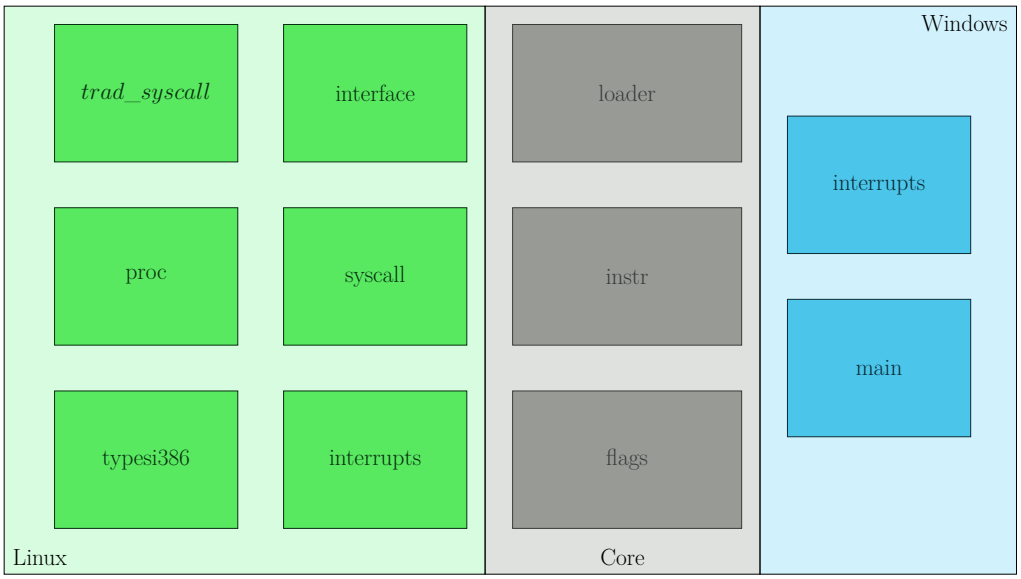

Figura D.1: Dependencias compartidas y propias entre ambos entornos

D.2. Versión para sistemas basados en Linux

Esta versión corresponde a la rama `main` del Repositorio del proyecto, que es aquella orientada a la ejecución en sistemas operativos basados en Linux, desde la terminal. Está pensada para ejecutarse de forma nativa en arquitecturas x86_64, de manera que tenga acceso a las llamadas del sistema, aunque puede funcionar (sin ellas) en otras arquitecturas.

Estructura de directorios

Si clonamos el repositorio, en concreto la rama `main`, obtendremos un directorio con el siguiente contenido.

- **Directorio lib**: contiene los archivos de cabecera (extensión .h).
- **Directorio src**: contiene los archivos fuente (extensión .c).
- **Directorio asms**: contiene ficheros con programas de ejemplo en ensamblador. Tanto código fuente como compilados a binario.
- **Directorio PoC**: contiene pruebas de concepto de vulnerabilidades y explotaciones en ensamblador.

- **Directorio debug**: contiene scripts y programas utilizados para las pruebas automáticas del simulador.
- **README.md**: fichero que contiene una breve descripción y explicaciones acerca de su instalación y uso.
- **LICENSE**: fichero que contiene el texto de la licencia del simulador.
- **NOTICE**: fichero que contiene los textos de las licencias de los otros proyectos utilizados.
- **compile.sh**: script que compila el ejecutable resultante.

Manual del programador

A continuación se va a proceder a detallar el funcionamiento del simulador, fichero a fichero y función a función, con la finalidad de documentar y poder comprender el código, de cara a futuras aportaciones de terceros.

flags.c

Define la variable global que simula el registro EFLAGS e implementa las funciones para testear, encender y apagar banderas, así como las funciones que determinan si una bandera debe ser encendida o no a partir de los operandos y el resultado de una operación. También otorga al registro EFLAGS su valor por defecto, que es el que debe tener al inicio del programa, cuando el loader ha acabado de cargar el programa y el registro EIP apunta a `_start`.

```
uint32_t eflags = 0x202;
```

Este valor corresponde al segundo bit encendido siempre por defecto, ya que es un bit reservado de Intel, y el bit de las interrupciones (IF), que es establecido por el kernel antes de devolver el control de la ejecución al usuario justo antes de cargar el programa en memoria y preparar su ejecución.

Las funciones que manipulan las banderas tienen la siguiente sintaxis.

```
void set_Flag(uint32_t f){ eflags |= (f); }
```

```
void clear_Flag(uint32_t f){ eflags &= ~(f); }

void complement_Flag(uint32_t f){ eflags ^= (f); }

int test_Flag(uint32_t f){ return (eflags & f)?1:0; }
```

Estas funciones se usan de manera que, se les pasa un valor que actúa como máscara para el registro EFLAGS, y con dicha máscara, se puede encender, apagar, complementar o testear una bandera específica. Dichas máscaras están definidas en el fichero `flags.h` como macros para el compilador, y pueden ser llamadas desde el código.

```
#define CF   0b0000000000000001 /* bit 0     */
#define PF   0b0000000000000100 /* bit 2     */
#define AF   0b0000000000010000 /* bit 4     */
#define ZF   0b0000000001000000 /* bit 6     */
#define SF   0b0000000010000000 /* bit 7     */
#define TF   0b0000000100000000 /* bit 8     */
#define IF   0b0000001000000000 /* bit 9     */
#define DF   0b0000010000000000 /* bit 10    */
#define OF   0b0000100000000000 /* bit 11    */
```

Por lo tanto, se pueden usar estos macros como argumentos en las funciones mencionadas anteriormente, de manera que por dentro harán la operación específica sobre el bit al que corresponden, dejando el resto del registro igual.

```
set_Flag(CF);   // eflags |= 0b0000000000000001
clear_Flag(CF); // eflags &= 0b1111111111111110
```

De igual manera, hay otras funciones que sirven para comprobar si, después de una operación específica, debe activarse una determinada bandera o no, en función de los operandos y el resultado. Estas funciones devuelven 1 si la bandera debe ser activada, o 0 si no. Algunas de estas funciones también necesitan la base de la operación, es decir, si los operandos son de

8, 16 o 32 bits. Cabe recalcar que la función que determina si la bandera OF (*overflow*) es calculada de manera distinta en función de si la operación es de suma o de resta.

```
int parity(uint32_t vv);
int zero(uint32_t v);
int sign(uint32_t v, uint8_t base);
int overflow_plus(uint32_t op1, uint32_t op2,
        uint32_t res, uint8_t base);
int overflow_minus(uint32_t op1, uint32_t op2,
        uint32_t res, uint8_t base);
int adjust(uint32_t op1, uint32_t op2, uint32_t res);
```

Para obtener más información sobre las banderas (*flags*), a qué corresponde cada una y cuando deben encenderse o apagarse, puede resultar útil la lectura del Apéndice C del manual de referencia del MIT sobre el i386 [4]. Este ha sido el manual consultado durante el desarrollo del proyecto.

typesi386.c

Implementa algunas funciones para la traducción de structs de 32 bits a 64 bits, para utilizarlos como argumento de las llamadas al sistema. Es llamado por `syscall.c`. Este fichero no tiene demasiada importancia, por lo que no de va a explicar en profundidad.

syscall.c

Define constantes del heap necesarias para alguna llamada al sistema, e implementa las funciones de las llamadas al sistema, donde algunas son traducciones de argumentos, y otras simulaciones de la llamada al sistema en cuestión. Todas las funciones de llamadas al sistema implementan la misma signatura, con el número de la llamada al sistema y 6 argumentos. Dichas funciones son llamadas por `trad_syscall.c`

En primer lugar, definimos algunos macros (que definen el inicio y final del HEAP simulado disponible) y variables para simular el heap a nivel funcional, sin simularlo profundamente.

```
#define HEAP_START 0x08060000
#define HEAP_LIMIT 0x09000000
```

```
static uint32_t sim_brk = HEAP_START;
```

A continuación, tenemos las funciones que implementan las llamadas al sistema. Todas tienen la misma signatura, devuelven un valor de tipo `uint32_t`, su nombre se compone de `do_` más el nombre de la llamada al sistema, y toman 7 argumentos, el número de la llamada al sistema (corresponde al valor del registro EAX) y sus 6 parámetros (corresponde a los valores de EBX, ECX, EDX, ESI, EDI, EBP). Dicha signatura es la siguiente.

También llevan asociado comentarios de documentación que hacen referencia a la llamada del sistema. Su nombre, lo que hacen, y como se pueden ejecutar mediante código C de 64 bits.

```
/**
 * brk, sbrk - change data segment size
 *
 * Syscall number 12
 *
 * int brk(void *addr);
 */
uint32_t do_brk(uint32_t *nr, uint32_t *arg1, uint32_t
            *arg2, uint32_t *arg3, uint32_t *arg4,
            uint32_t *arg5, uint32_t *arg6){
    ...
}
```

La implementación de estas llamadas al sistema ha sido parcial. Se han implementado aquellas necesarias para ciertas demostraciones de ciberseguridad o para el correcto funcionamiento de un programa ELF-32 en el simulador. Por su complejidad, han quedado llamadas al sistema sin implementar.

trad_syscall.c

Define un vector de punteros a funciones de llamadas al sistema, donde la posición de cada función en el vector corresponde a su número de llamada al sistema en el i386. Es decir, la *syscall* `exit`, con número 1, ocupa la segunda

posición del vector (índice 1, ya que la primera posición es índice 0). Es llamado por `interrupts.c`.

Para ello, en primer lugar se ha definido el tipo `Syscall32bits` en el fichero `trad_syscall.h`. Este tipo corresponde a un puntero a función, donde la función tiene la misma signatura que las funciones que implementan las llamadas al sistema mencionadas anteriormente.

```
typedef uint32_t(*Syscall32bits)(uint32_t *nr, uint32_t
            *arg1, uint32_t *arg2, uint32_t *arg3, uint32_t
                *arg4, uint32_t *arg5, uint32_t *arg6);
```

Por ello, este tipo es un *wrapper* para poder almacenar punteros a llamadas al sistema en un array, que veremos definido en el fichero `trad_syscall.c`.

```
Syscall32bits syscalls[]= {
    do_restart_syscall, do_exit, do_fork, ...
    unimpl, ...
    ...
}
```

interrupts.c

Implementa la gestión de interrupciones, que se limita a las llamadas al sistema. Es usado por `instr.c`. Para ello implementa la IDT, que es la tabla de interrupciones. Dicha tabla se comprende como un array de un tipo personalizado, donde cada elemento del array son punteros a funciones.

```
typedef uint32_t (*Interrupt)(uint32_t *eax, uint32_t *ebx,
    uint32_t *ecx, uint32_t *edx, uint32_t *esi, uint32_t *edi,
    uint32_t *ebp);
```

Por ello, con este tipo podemos definir un vector donde cada posición corresponda al *handler* de la interrupción, de manera que si hacemos `idt[0x80](eax, ebx,...)`, ejecute el handler de la interrupción 0x80, que es la llamada al sistema.

```
Interrupt idt[] = { unimplemented, unimplemented, ...
                    int80        , unimplemented, ...
                  }
```

Una vez definida la IDT, se ha implementado una función que servirá como intermediario entre la ejecución de las interrupciones y la ejecución de las instrucciones. Dicha función se llama `int_dispatcher` y será llamada por el fichero `instr.c` en instrucciones como INT. El parámetro más importante de esta función es el primero, el valor `v`, que es un valor del 0 al 255 y que indica la interrupción a manejar.

```
uint32_t int_dispatcher(uint8_t v, uint32_t *eax, uint32_t
    *ebx, uint32_t *ecx, uint32_t *edx, uint32_t *esi,
    uint32_t *edi, uint32_t *ebp){

    return  idt[v](eax, ebx, ecx, edx, esi, edi, ebp);
}
```

Gran parte de las interrupciones no han sido implementadas ya que son interrupciones de error que finalizarían el programa y no son útiles para la simulación del programa. La interrupción más importante es la 0x80, que ha sido implementada de la siguiente forma, aunque primero es necesario redefinir (usando la palabra reservada `extern`) la tabla de syscalls, para obtener una referencia a la definida en el fichero `trad_syscall.c`.

```
extern Syscall32bits syscalls[];

uint32_t int80(uint32_t *eax, uint32_t *ebx, uint32_t
    *ecx, uint32_t *edx, uint32_t *esi, uint32_t *edi,
    uint32_t *ebp){

    *eax = syscalls[*eax](eax, ebx, ecx, edx,
                          esi, edi, ebp);
    return *eax;
}
```

En la función `int80`, llamamos a la función definida por el puntero que se encuentre en la posición correspondiente al valor del registro EAX, y una vez finalice dicha función, guardaremos el resultado que devuelva en el propio registro EAX. De igual manera, esta función también devolverá el valor que se acaba de almacenar en el registro EAX.

loader.c

Implementa el cargador del programa, que lee el fichero ejecutable, lo carga en memoria, y prepara la pila antes de pasarle el control al usuario. Simula la tarea del kernel de preparar la pila (valores auxiliares, variables de entorno, argumentos, *padding*) si el modo elegido es el modo ELF.

Si el modo es, efectivamente, el modo ELF, realiza las siguientes tareas.

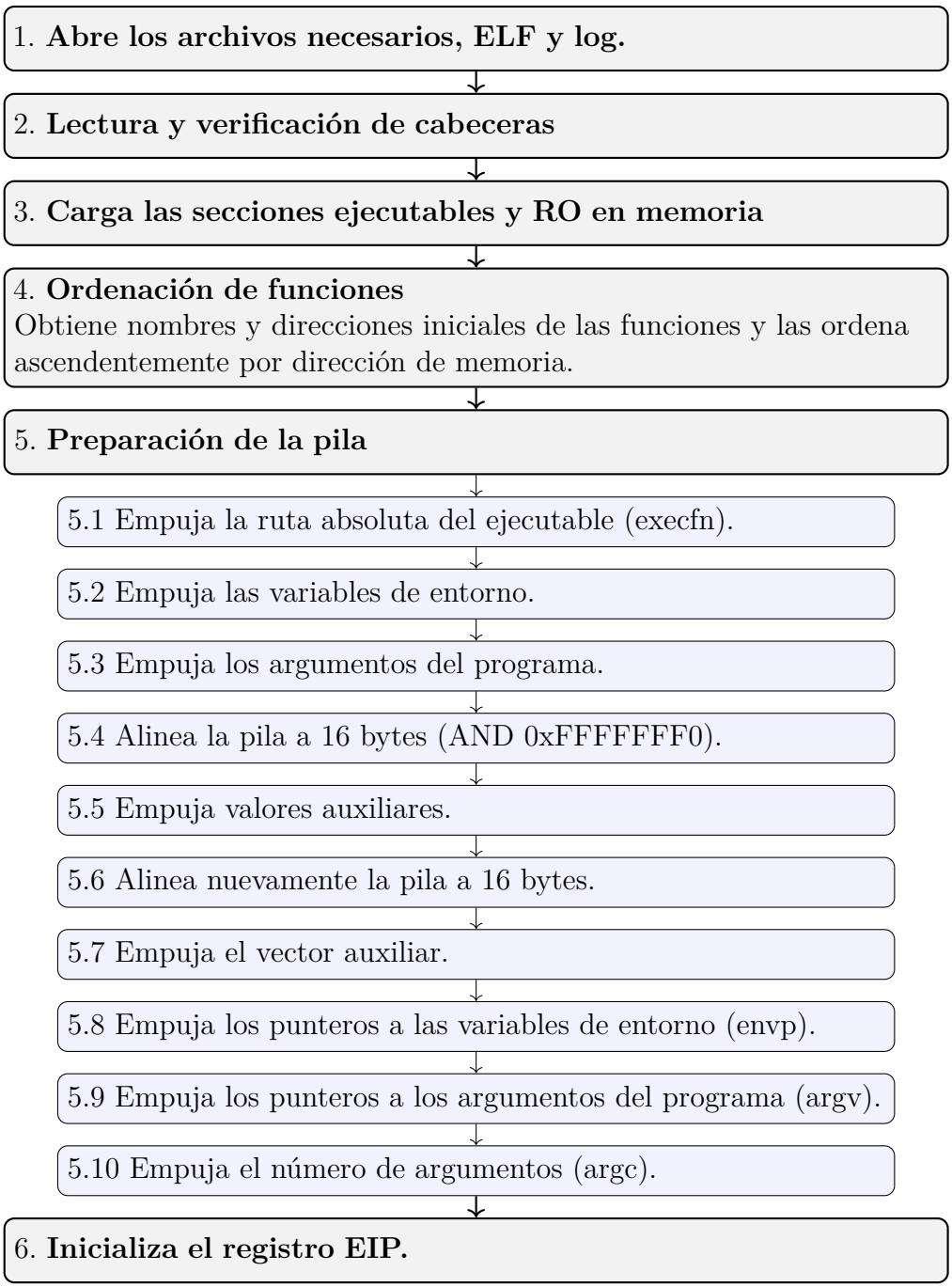

Figura D.2: Proceso completo de carga, preparación de memoria e inicialización del programa

Modo ELF En el modo ELF, se ejecuta la función `read_elf_file`, que toma varios argumentos.

Tipo y nombre	**Inicialización**	**Explicación**
int argc	Fuera	Número de argumentos del programa
char * argv[]	Fuera	Array de punteros a los argumentos del programa
char * envp[]	Fuera	Array de punteros a las variables de entorno
uint32_t ** sheader	Dentro	Array que contiene dos posiciones por cada sección ejecutable (Dirección de memoria y tamaño)
uint32_t * count	Dentro	Número de secciones ejecutables
uint32_t ** symbols	Dentro	Array que contiene las direcciones de memoria de las funciones
char ** strtab	Dentro	Array que contiene las cadenas con los nombres de las funciones (sus índices coinciden con symbols)
uint32_t * ccc	Dentro	Número de funciones del programa

Tabla D.1: Argumentos de la función loader para ficheros ELF.

Las variables cuya inicialización se indica como "Dentro"de la función, indican que dichas variables o bien se les reserva memoria dentro de la función, que se les asigna un valor dentro de la función para devolvérselo a la función que la llama, o ambas. Por ejemplo, en el caso de los dobles punteros `symbols` y `strtab`, reservan memoria dentro de la función mediante el uso de `calloc`, y a continuación se les asignan sus valores correspondientes, ya que son usados en el fichero `proc.c`, durante `interface_elf_main`.

En primer lugar se abre el fichero log, por si hubiera algún error, para poder reflejarlo. A continuación se abre el fichero ELF a ejecutar, que corresponde al argumento `argv[0]`, y el modo es de lectura binaria.

```
FILE *elf_file = fopen(argv[0], "rb");
if (!elf_file) {
    perror("Error al abrir ELF");
    puts("Failed to open ELF file.");
    fprintf(log, "Failed to open ELF file. \n");
    return 1;
```

```
}
```

Una vez abierto el fichero, y comprobado que el puntero a fichero sea válido, se lee la cabecera ELF del fichero, de la siguiente manera.

```
Elf32_Ehdr ehdr;
fread(&ehdr, 1, sizeof(ehdr), elf_file);
fprintf(log, "Read ELF headers. \n");
```

De esta manera creamos un tipo estructurado que corresponde a la cabecera del archivo y almacenamos en él la información que leamos del fichero. A continuación, comprobamos la clase del fichero. Los primeros 4 bytes del fichero (conocidos como *magic*) determinan si es un fichero ELF o no, y además se comprueba otra flag adicional para asegurarnos de que además de ser un fichero ELF, es uno de 32 bits. Por ello vemos que primero comprueba si coinciden los bytes *magic*, y luego comprueba que la clase sea de 32 bits.

```
if (memcmp(ehdr.e_ident, ELFMAG, SELFMAG) != 0 ||
    ehdr.e_ident[EI_CLASS] != ELFCLASS32) {
    fprintf(stderr, "Archivo no es ELF32 válido.\n");
    fprintf(log, "Invalid ELF file type. \n");
    fclose(elf_file);
    return 1;
}
```

A continuación se valida que el puntero a memoria (`if (!mem)...`) sea válido, ya que a partir de aquí se va a cargar información en memoria. Pero primero, se deben leer los *Program Headers* del fichero binario.

```
fseek(elf_file, ehdr.e_phoff, SEEK_SET);
uint8_t found = 0;
long phdrs_start = ehdr.e_phoff;
long shdrs_start = ehdr.e_shoff;

/* Values for pushing auxv */
```

```
ph_num = ehdr.e_phnum;
ph_size = ehdr.e_phentsize;

uint32_t progbits[2];
uint8_t progc = 0;
```

Con esto avanzamos en el puntero al archivo hasta donde comienzan los *program headers*, y también almacenamos donde comienzan los *section headers*. De igual manera, almacenaremos otros valores como el número de *program headers* y su tamaño en bytes, ya que lo necesitaremos más adelante durante la preparación de la pila. Las ultimas dos variables se utilizan para almacenar las direcciones de memoria de las secciones ejecutables (`PROGBITS`).

Si `progc` es 0 y nos encontramos una de estas secciones, la almacenamos en la posición 0 y seguimos buscando. Cuando `progc` ya es 1, seguimos almacenando direcciones de memoria en la posición 1 hasta llegar a la última. Esto hace que una vez hemos recorrido todas las secciones, lo que tengamos almacenado sea el principio y el fin de las secciones ejecutables.

A continuación, iteraremos sobre los *program headers* buscando aquellos que tengan la flag `PT_LOAD`, ya que significa que deben ser cargados en memoria. Si no tienen esa flag en su `phdr.p_type`, podremos saltar esa iteración mediante la sentencia `continue;`.

```
for (int i = 0; i < ehdr.e_phnum; i++) {
    fseek(elf_file, phdrs_start + i * sizeof(Elf32_Phdr),
            SEEK_SET);
    Elf32_Phdr phdr;
    fread(&phdr, 1, sizeof(phdr), elf_file);

    if (phdr.p_type != PT_LOAD)
        continue;
```

Una vez estamos en una iteración que sí que corresponde a un *program header* cargable, lo copiamos a memoria usando la función `fread`, ya que tenemos el tamaño del segmento en `phdr.p_filesz`. Y, a continuación, hacemos la búsqueda de `PROGBITS` que hemos mencionad anteriormente.

```
        fseek(elf_file, phdr.p_offset, SEEK_SET);
        if (!i){
            ph_addr = phdr.p_vaddr + ehdr.e_phoff;
        }
        fread(mem + phdr.p_vaddr, 1, phdr.p_filesz,
                            elf_file);

        if(!progc){
            /* Save first and PROGBITS section*/
            progbits[progc]=phdr.p_vaddr;
            progc++;
        }else{
            /* Rewrite progbits[1] until
            obtains the last one */
            progbits[progc]=phdr.p_vaddr+phdr.p_filesz;
        }
        fprintf(log, "Segment loaded. vaddr=0x%08x,
            size=%u bytes\n" ,phdr.p_vaddr, phdr.p_memsz);
    }
```

A continuación, crearemos un array de tamaño igual al doble del número de *section headers*, para almacenar en él cada par de dirección de memoria y tamaño. De esta manera, en la posición `2*i` tendremos la dirección de memoria de la sección, y en la posición `2*i+1` tendremos su tamaño. También inicializaremos la cuenta de secciones ejecutables a 0, y crearemos un array para guardar los *section headers* de `SYMTAB` y de `STRTAB`, que contienen la información sobre las funciones, y sus nombres, respectivamente.

```
*sheader = calloc(ehdr.e_shnum*2, sizeof(uint32_t));
*count = 0;

/* Variable for storing SYMTAB([0]) and STRTAB([1]) */
Elf32_Shdr tabs[2];

/* Iterate Section Headers */
for (int i = 0; i < ehdr.e_shnum; i++) {
    fseek(elf_file, shdrs_start + i * ehdr.e_shentsize,
                        SEEK_SET);
    Elf32_Shdr shdr;
```

```
        fread(&shdr, 1, sizeof(shdr), elf_file);
        if (shdr.sh_type == SHT_PROGBITS
                && (shdr.sh_flags & SHF_EXECINSTR)) {
            (*sheader)[2* (*count)]   = shdr.sh_addr;
            (*sheader)[2*(*count)+1] = shdr.sh_size;

            *count += 1;
        }else if(shdr.sh_type == SHT_SYMTAB){
            tabs[0] = shdr;
        }else if(shdr.sh_type == SHT_STRTAB
                            && i != ehdr.e_shstrndx){
            /* We want .strtab, not .shstrtab */
            tabs[1] = shdr;
        }

    }
```

En este bucle, buscaremos las secciones que sean `PROGBITS` y con flag de ejecutable, para posteriormente almacenarlas (junto con su tamaño). Al final del bucle tendremos en el array `sheader` la información de todas las secciones ejecutables (para poder llevar a cabo su desensamblaje posteriormente en el fichero `proc.c`), en la variable `count` tendremos el número de secciones ejecutables, es decir, el **tamaño real** (usado) del array `sheader` si lo multiplicamos por 2, y los *section headers* con lo necesario para extraer los nombres de las funciones y sus direcciones de memoria.

A continuación, comprobamos que los argumentos `symbols` y `strtab` contienen basura, para poder utilizarlos ahora. Es una manera de poder indicar desde `proc.c` si se deben obtener los nombres de las funciones o no.

```
    /* If ELF type of execution, they will contain trash,
            but not NULL */
    if (symbols != NULL && *strtab != NULL){

        /* Number of symbols in SYMTAB */
        int nsyms = tabs[0].sh_size / sizeof(Elf32_Sym);

        /* Allocate memory for the entire strtab */
        *strtab = calloc(sizeof(char), tabs[1].sh_size +1);
```

```
/* Move pointer to the start of strtab in the file*/
fseek(elf_file, tabs[1].sh_offset, SEEK_SET);
/* Read strtab to the local pointer variable */
fread(*strtab, tabs[1].sh_size,1, elf_file);

/* Move file pointer to first symbol in SYMTAB */
fseek(elf_file, tabs[0].sh_offset, SEEK_SET);

/* Allocate space for storing each symbol's name
        and value */
*symbols = calloc(nsyms*2 + 1, sizeof(uint32_t));

/* Counter within array. We are not going
            to use it entirely.
    Some symbols are not functions. */
*ccc = 0;
```

Inicializamos las variables que sirven para lo siguiente. La variable `nsyms` contiene el número de símbolos que hay en la `symtab`, aunque no todos ellos corresponden a funciones. La variable `strtab` es inicializada con la función `calloc` para reservar memoria para todos los strings. Luego inicializamos la variable `ccc`, que va a llevar la cuenta de los símbolos que sí que son funciones, y por lo tanto almacenaremos su nombre. La variable `symbols` es inicializada con `calloc` para reservar memoria para cada par de dirección de memoria de la función y puntero a string.

A continuación, iteramos sobre todos los símbolos para averiguar cuales de ellos son funciones, y por lo tanto almacenarlos.

```
/* Iterate through Symbol table entries */
    for(int i=0; i< nsyms; i++){
        Elf32_Sym sym;
        fread(&sym, sizeof(Elf32_Sym), 1, elf_file);
        int type = ELF32_ST_TYPE(sym.st_info);
        if (type == STT_FUNC || (type == STT_NOTYPE &&
                sym.st_value >= progbits[0] &&
                sym.st_value <= progbits[1])){
                /* Is FUNC or NOTYPE within .text */
            /* Get string and store it */
```

```
            /* strtab Index */
            (*symbols)[*ccc*2]   = sym.st_name;
            /* vaddr */
            (*symbols)[*ccc*2+1] = sym.st_value;
            (*ccc)++;
        }
    }
```

Para averiguar si un símbolo es una función, su tipo debe ser `STT_FUNC`, o ser `STT_NOTYPE` y que su dirección de memoria esté comprendida entre los dos valores que hemos almacenado en el array `progbits`. Si cumple alguna de las dos condiciones, guardamos su nombre, su dirección de memoria, y aumentamos la variable `ccc`.

A continuación, una vez tenemos todos los nombres y direcciones de las funciones (que son las funciones que luego vamos a ver como `_start`, `main` o `printf`), vamos a proceder a ordenarlas mediante un algoritmo *Bubble Sort*, para que el tiempo de búsqueda a la hora de imprimirlas se reduzca. Es decir, que cuando vayamos a imprimir la interfaz, y tengamos que buscar a qué función pertenece una instrucción en una determinada dirección de memoria, el tiempo de búsqueda sea menor.

```
/* Sort vaddrs */
for(int i=0; i<(*ccc)-1; i++){
    int min_idx = i;
    for (int j = i + 1; j < *ccc; j++) {
        if ((*symbols)[2*j+1] < (*symbols)[2*min_idx+1]) {
            min_idx = j;
        }
    }
    if(min_idx != i){   /* Exchange */
        uint32_t temp  = (*symbols)[i*2+1]; /* vaddr*/
        uint32_t temp1 = (*symbols)[i*2]; /* index */

        (*symbols)[i*2+1] = (*symbols)[min_idx*2+1];
        (*symbols)[i*2] = (*symbols)[min_idx*2];

        (*symbols)[min_idx*2+1] = temp;
        (*symbols)[min_idx*2]   = temp1;
    }
```

```
}
```

Por último, una vez hemos cargado el ejecutable en memoria, y hemos almacenado la información sobre las funciones para usarlas a la hora de imprimir la interfaz, cerraremos los dos ficheros que hemos usado, el fichero ELF y el fichero log, inicializaremos el registro EIP (ya que su valor viene dado por el *entry point* en el fichero ELF y cargaremos la pila.

```
fclose(elf_file);
fclose(log);

/* EIP Initialization */
eip = ehdr.e_entry;

load_stack(argc, argv, envp);

/* No need to free mem pointer,
    because it is used in the main program */

return 0;
```

Una vez cargada la pila, salimos de la función y es como si el kernel hubiera devuelto el control al usuario tras cargar el programa.

Modo RAW En cuanto al modo del *loader* que carga programas sin cabeceras, se ejecuta la función `read_raw_file`, que toma los siguientes argumentos.

Tipo y nombre	**Inicialización**	**Explicación**
int argc	Fuera	Número de argumentos del programa
char * argv[]	Fuera	Array de punteros a los argumentos del programa
char * envp[]	Fuera	Array de punteros a las variables de entorno
uint32_t * count	Dentro	Número de bytes cargados

Tabla D.2: Argumentos de la función loader para ficheros RAW.

En primer lugar, al igual que en el modo ELF, realizamos las aperturas de los ficheros. Abriremos tanto el fichero *log* como el fichero ejecutable sin cabeceras (*RAW*).

```
/* LOG File Initialization */
FILE *log = fopen("log.txt", "w");
if (log == NULL) {
    perror("Error al abrir el archivo de log");
    return 1;
}

/* Open raw file to perform binary read */
/* argc was already checked at proc.c so it
        must be at least 3 */
FILE *raw_file = fopen(argv[0], "rb");
if (!raw_file) {
    perror("Error al abrir raw file");
    puts("Failed to open raw file.");
    fprintf(log, "Failed to open raw file. \n");
    return 1;
}
```

A continuación, como no tenemos cabeceras que nos indiquen el tamaño de la sección ejecutable, debemos obtener el tamaño en bytes del fichero, que lo haremos llevando el puntero hasta el final del fichero, (`SEEK_END`), y usando la función `ftell` para que nos indique en qué byte estamos. Luego utilizaremos la función `rewind` para volver al inicio del archivo, para empezar a cargarlo.

```
/* Obtain file's size */
fseek(raw_file, 0, SEEK_END);
size_t size = ftell(raw_file);

rewind(raw_file);
```

Para cargar el archivo en memoria, ya que el archivo completo es una sección ejecutable, podemos usar una función como `memcpy`.

```
size_t readbytes = fread(mem, 1, size, raw_file);
if (readbytes != size) {
    perror("Error al leer el archivo completo");
    fprintf(log, "Failed to open raw file. \n");
    fclose(log);
    fclose(raw_file);
    return EXIT_FAILURE;
}
```

Por último, inicializamos el registro EIP (a 0 ya que la primera instrucción está en el byte 0 y ha sido cargada en `0x00000000`), y cerramos los ficheros que acabamos de utilizar. También inicializamos el parámetro `count` que ha sido pasado por referencia, para que contenga el número de bytes del fichero, ya que puede ser útil para usarlo en `proc.c` durante la ejecución principal.

```
*count = size;

eip = 0x00000000;

fclose(raw_file);
fclose(log);

return 0;
```

interface.c

Es el fichero fuente que implementa las funciones necesarias para mostrar el flujo de ejecución del simulador en una interfaz gráfica integrada en la terminal. Inicialmente, la interfaz iba a ser desarrollada con la librería `ncurses` de C, pero tras una fase de análisis, se decidió utilizar una interfaz de implementación propia para conseguir mayor flexibilidad en cuanto a la impresión de cadenas, necesarias a la hora de implementar las funcionalidades del simulador.

La interfaz funciona de manera que, por norma general, es puesta en modo *raw*, es decir, los bytes equivalentes a las teclas pulsadas por el usuario son enviadas al buffer de entrada de manera inmediata, sin tener que esperar a que el usuario presione la tecla `ENTER`, que equivale a `'\n'` (o byte `0x0A`). Este modo de la terminal consigue que, cuando el usuario indica una opción,

por ejemplo la tecla `'S'`, no tenga que darle a `ENTER` para que la selección sea efectiva. De igual manera, tampoco era factible utilizar un `getchar()` habitual de la *libc* junto con la terminal en modo canónico (el normal), ya que teclas como las flechas de dirección equivalen a 3 bytes, y la función `getchar()` solo lee uno a la vez.

Para ello, primero hemos definido una serie de variables globales correspondientes al `struct termios`, que sirven como una especie de *caché* de cuál es el estado actual de la terminal (si está en modo *raw* o canónico), y cuál es el estado anterior, por si hubiera que restablecer alguno de ellos.

```
struct termios oldt, newt, olde, newe;
```

Modificamos el estado de la terminal mediante las funciones `init_raw_mode()`, `enable_raw_mode()` y `disable_raw_mode()`, que inicializan, activan y desactivan el modo raw de la terminal, respectivamente. Desactivar el modo *raw* de la terminal equivale a ponerla de nuevo en modo canónico.

```
void init_raw_mode(){

    tcgetattr(STDIN_FILENO, &oldt);
    newt = oldt;
    // Sin modo canónico y sin eco (~(ICANON | ECHO))
    newt.c_lflag &= ~(ICANON);
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
}

void enable_raw_mode() {
    // Guarda la config actual
    newt = oldt;
    // Sin modo canónico y sin echo (~(ICANON | ECHO))
    newt.c_lflag &= ~(ICANON);
    tcsetattr(STDIN_FILENO, TCSANOW, &newt);
}

void disable_raw_mode() {
    // Restaura configuración antigua
    tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
}
```

También se han definido funciones para restablecer el modo *echo* de la terminal (mostrar la entrada del usuario o no), ya que para ciertas opciones de la interfaz, como escribir direcciones de memoria para poner un breakpoint, es necesario que el usuario vea su input.

```
void disable_echo(){
    // Guarda la config actual
    tcgetattr(STDIN_FILENO, &olde);
    newe = olde;
    newe.c_lflag &= ~(ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &newe);
}

void enable_echo(){
    newe = olde;
    newe.c_lflag &= ~(ECHO);
    tcsetattr(STDIN_FILENO, TCSANOW, &newe);
}
```

Una vez configurado el modo de la terminal, está preparado para mostrar la interfaz. Tal como se ha comentado anteriormente, la interfaz está pensada para tener 4 ventanas o secciones. Una sección superior diferenciada que muestre los registros, dos ventanas laterales diferenciadas que contengan el código (las instrucciones) y la pila, y una última ventana inferior reservada para la salida del programa (por si hace `write()`) o para las opciones que puede pedir la interfaz gráfica. Por ello, se definen ciertas constantes para dicha interfaz. Estas constantes están definidas en el fichero `lib/interface.h`.

```
#define H_REGS 8
#define W_STACK 27
#define H_CMD 5
```

Estas constantes indican lo siguiente:

- `H_REGS` define la altura (*Heigth*) que debe tener la ventana de registros, que corresponde a 8 líneas. 2 de ellas están reservadas para el recuadro

que envuelve la ventana (línea superior e inferior) y las otras 6 líneas para mostrar los 10 registros y los 6 selectores de segmentos, mostrando 3 cada línea.

- `W_STACK` define el ancho (*Width*) de la ventana de la pila. Equivale a 27 ya que deben haber dos valores en hexadecimal de longitud 10 cada uno (`0x12345678`), 2 caracteres reservados para el recuadro que envuelve la ventana, un caracter reservado para el `":"`que separa los valores hexadecimales, y 4 espacios para separar el recuadro de los valores, y los valores del `":"`.
- `H_CMD` define la altura (*Heigth*) que debe tener la ventana de CMD. La altura seleccionada ha sido de 5 líneas para que cualquier valor a imprimir pueda ser impreso sin problema.

A continuación, para construir los recuadros que envuelven las ventanas, se ha creado una función que devuelve una cadena de la longitud `n` que se le especifique, conteniendo únicamente el caracter `"-"` repetido `n` veces, con el objetivo de crear una línea para el recuadro.

```
void get_lines(size_t size, char * str){
    const char *hline = "-";

    for (int i=0; i<size; i++){
        strcat(str, hline);
    }
}
```

La función inicial del fichero `interface.c` es la función `init_interface()`, que es llamada por `proc.c` al iniciar el simulador. Esta función, en primer lugar, utiliza el *wrapper* de la *libc* para la llamada al sistema `ioctl`, para obtener el alto y ancho (en lineas y caracteres) de la terminal en la que se está ejecutando el programa. Estos valores son importantes a la hora de construir la interfaz.

```
    struct winsize w;

    if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &w) == -1) {
        perror("ioctl");
```

```
    return 1;
}
/* Get terminal size */
rows = w.ws_row;
cols = w.ws_col;
```

Una vez tenemos las medidas de la terminal, calculamos los espacios de separación que va a haber entre cada registro en la ventana de registros. Cada registro ocupa 27 caracteres y hay 3 por línea, y también debemos tener en cuenta que hay 4 caracteres reservados, 2 para el recuadro y otros 2 para espacios entre el recuadro y el registro contiguo. Por último, el espacio restante se va a dividir entre dos sitios, ya que al haber 3 registros, hay 2 separaciones entre ellos. Si el espacio es demasiado grande, lo reduciremos para que no se dispersen demasiado.

```
/* Get spaces between registers */
spaces = ((cols - 4)-(27*3))/2;
if(spaces)spaces--;
if (spaces > 15)spaces=10;
```

A continuación, reservaremos espacio para las variables que van a contener las líneas de los recuadros de cada ventana. Reservamos posiciones de 3 bytes ya que los caracteres que componen los recuadros son caracteres especiales Unicode y ocupan 3 bytes cada uno. Reservamos `n-1` posiciones ya que del total de ancho de una línea, 2 caracteres son de esquinas así que no los incluiremos, por lo que deberíamos reservar `n-2` posiciones, pero como los *strings* deben acabar en `0x00`, reservaremos `n-1`.

```
/* Alloc memory for drawing the boxes */
lines = calloc(3, cols-1);
code = calloc(3,w_code-1);
stack = calloc(3, w_stack-1);
/* Fill code and stack strings with box lines */
get_lines(cols-2, lines);
get_lines(w_code-2, code);
get_lines(w_stack-2, stack);
```

La variable `lines` va a contener las líneas de la ventana de registro, `code` va a contener las líneas para la ventana de código, y `stack` contendrá las líneas para la ventana de pila. Todas estas variables excluyen las esquinas para poder ser usadas tanto en el borde superior como en el borde inferior.

Por último, almacenaremos los valores actuales de los registros, para que, la próxima vez que imprimamos la interfaz, podamos indicar los registros que han cambiado y resaltarlos.

```
/* Store old regs values to compare and highlight if change */
old_eax = eax;
old_ebx = ebx;
old_ecx = ecx;
old_edx = edx;
old_ebp = ebp;
old_esp = esp;
old_esi = esi;
old_edi = edi;
old_eflags = eflags;
```

Para ello, hemos definido referencias a las variables de los registros definidas en `instr.c` y `flags.c`, y creado nuestras propias variables globales en `interface.c` para almacenar dichos valores.

```
extern uint32_t eax, edx, esp, esi, eip, cs, ds,
    fs, ecx, ebx, ebp, edi, ss, es, gs;
uint32_t old_eax, old_edx, old_esp, old_esi,
    old_ecx, old_ebx, old_ebp, old_edi, old_eflags;
```

Posteriormente, como alternativa a `getchar()`, que ya hemos dicho que no nos sirve porque lee solamente un byte y hay teclas que se traducen como varios bytes, se ha creado la función `getch()`, que en vez de obtener un `char` (1 *byte*), obtiene un caracter completo. Esto nos permite detectar tanto las teclas convencionales como las flechas de dirección.

A continuación se han creado una serie de funciones auxiliares que permiten mover el cursor (la posición en terminal del siguiente caracter a imprimir) o borrar líneas. Estas funciones se usan de manera complementaria

a la hora de imprimir la interfaz y se llaman `move` o `clean`. Podemos verlas por ejemplo, en la función principal de la interfaz, `draw_screen()`, que toma los siguientes argumentos.

Tipo y nombre	Inicialización	Explicación
int scr_s	Fuera	Número de doublewords de diferencia entre el ESP y el primer valor del Stack a imprimir. Si es 0, el primer valor que se imprime es el ESP, si es 1, ESP + 4, etc.
int scr_c	Fuera	Número de instrucciones de diferencia entre la primera de todas las existentes y la primera instrucción a imprimir.
char ** lineas	Fuera	Array que contiene para cada instrucción i, su dirección de memoria en la posición 2*i, y su nemotécnico y argumentos en 2*i+1.
int count	Fuera	Número de líneas a imprimir en el código.
int eip_ind	Fuera	Índice de la instrucción que corresponde al EIP.
char **funcs	Fuera	Array que contiene las funciones de las instrucciones a imprimir.

Tabla D.3: Argumentos de la función draw_screen() de la interfaz.

Una vez en la función, lo primero que hacemos es limpiar la terminal completa, menos las últimas 3 líneas, para que si en la interación anterior el programa ha escrito algo, no lo borremos. Luego comprobamos los estados de las banderas y los almacenamos en variables `uint8_t`. También moveremos el cursor de nuevo a la primera línea, para proceder a imprimir la interfaz.

```
/* Clear screen */
cleanv(1, rows-3);

/* Test Flags to show on Registers Window */
uint8_t c = test_Flag(CF),p = test_Flag(PF),
    z = test_Flag(ZF),s = test_Flag(SF),
    o = test_Flag(OF),a = test_Flag(AF),
```

```
        i = test_Flag(IF);

    /* Set cursor */
    move(1);
```

Una vez tenemos la terminal limpia y el cursor está en la primera línea, imprimimos la ventana de los registros, donde emplearemos la variable `lines`. Primero imprimiremos la línea superior, con las dos esquinas y la variable `lines` en medio, y luego 6 líneas de registros que siguen la siguiente estructura.

- Placeholder para secuencia de escape de resalte.
- Nombre del registro.
- Placeholder para valor del registro en hexadecimal.
- Placeholder para valor del registro en decimal (algunos).
- Placeholder para cierre de secuencia de escape de resalte.
- Placeholder para separación con respecto del siguiente registro.

Los selectores de segmento y el registro EFLAGS son ligeramente distintos, pero en esencia funcionan de igual manera, con placeholders para poner los valores de los registros. El registro EIP no cuenta con placeholder para la secuencia de escape de resalte ya que se supone que debe cambiar casi siempre (excepto en instrucciones de tipo REP). Los selectores de segmento tampoco son resaltados.

```
printf("| %sEAX : 0x%08x %10d%s %*s
          %sECX : 0x%08x %10d%s %*s
          %sESI : 0x%08x %010u%s\n",
    eax!=old_eax?"\033[7m":"",eax, eax,"\033[0m" ,spaces, " ",
    ecx!=old_ecx?"\033[7m":"",ecx, ecx,"\033[0m" ,spaces, " ",
    esi!=old_esi?"\033[7m":"",esi, esi,"\033[0m");
/* Moves the cursor to the end of the terminal
    line and puts the closing character */
printf("\033[%d;%dH|\n", 2, cols);
```

A continuación, realizamos de nuevo la asignación de valor antiguo de los registros, para usarlos en la siguiente iteración.

```
/* Update old values */
old_eax = eax;
old_ebx = ebx;
old_ecx = ecx;
old_edx = edx;
old_ebp = ebp;
old_esi = esi;
old_edi = edi;
old_eflags = eflags;
```

Posteriormente, imprimiremos los bordes superiores de las ventanas de código y pila, utilizando las variables `code` y `stack`.

```
/* Code and Stack Box Top Line*/
printf("|-%*s-|-%*s-|\n",
        w_code-2, code, w_stack-2, stack);
```

Una vez impresa la línea superior, procedemos con las líneas de código. Dichas líneas son impresas en un bucle de `i=0` hasta `i=rows-H_REGS-5`, que equivale al total de líneas menos las reservadas para los registros menos las reservadas para la CMD.

```
/* Code */
for (int i=0; i<rows-H_REGS-5; i++){
    if (i < count){
        /* Clear line and Print code addr in blue */
        if(funcs != NULL){
            printf("\033[K| %s%s%s <%s>: ",
                "\033[34m", lineas[i*2], "\033[0m",
                                funcs[i]);
        }else{
            printf("\033[K| <%s%s%s>: ",
                "\033[34m", lineas[i*2], "\033[0m");
```

```
            }

            if(eip_ind >= 0 && eip_ind == i){
                /* Highlight current instruction (EIP) */
                printf("%s%s%s",
                    "\033[7m",lineas[i*2+1],"\033[0m");
            }else{
                /* Print normal instruction */
                printf("%s",
                    lineas[i*2+1]);
            }
            /* Close box line */
            printf("\033[%d;%dH|\n", H_REGS+i+2, w_code);
        }
    }
```

En este bucle, si la iteración actual es menor que el número de instrucciones a escribir (variable `count`), haremos lo siguiente. Primero imprimir la dirección de memoria, pero si la variable `funcs` es no nula (equivaldría a estar ejecutando un fichero ELF, ya que `funcs` es nula al ejecutar un fichero raw), se añade un placeholder adicional para `funcs[i]`. Posteriormente, si la instrucción coincide con el EIP, se resalta, y si no, se imprime normal, y por último, se rellenan caracteres hasta el final y se escribe el caracter de cierre de la línea.

Utilizamos el vector `lineas` de manera que en la posición `2*i` contiene la dirección de memoria de la instrucción `i`, y en `2*i+1` contiene la cadena de la instrucción en cuestión.

Una vez impreso el código, pasaremos a imprimir la pila. En primer lugar definiremos algunas variables auxiliares. La variable `ii` es el número de *doublewords* de diferencia, contando desde `ESP`, con el primer valor a imprimir. Si el primer valor es ESP, `ii` será 0, si el primer valor es ESP+8, `ii` será 2. La variable `k` es el contador de líneas ya impresas, `lim` es el límite de líneas que se pueden imprimir, para no hacer overflow a otras ventanas, y `j` es el número de valores en pila a imprimir sin que se salgan del rango de la pila.

```
    /* First position to draw from the stack
        (Starting i doublewords from ESP) */
```

```
        int ii = scr_s;
        /* Lines drawn counter */
        int k=0;
        /* Rows available to draw */
        int lim = rows - H_REGS - 2 - 3;
        /* Substraction with unsigned ints so
            we must cast to int */
        /* Limit of rows available to draw so
            it doesnt overflow STACK_BOTTOM */
        int j = ((int)(STACK_BOTTOM - esp))/4;
```

Usamos un bucle do-while, que parará si hemos impreso tantas líneas como podíamos (`k == lim`) o si no hay más valores de pila a imprimir (`ii == j`).

```
do{
    if (j > 0 && esp + 4*ii < STACK_BOTTOM){
        printf("\033[%d;%dH| %s0x%08x%s : 0x%08x \033[%d;%dH|\n",
            H_REGS+k+2, w_code+1, "\033[32m", esp+4*ii, "\033[0m",
            *((uint32_t *)(mem+(esp+4*ii))), H_REGS+k+2, cols);
    }
    if (esp + 4*ii >= STACK_BOTTOM){
        break;
    }
    ii++;
    k++;
}while(k < lim && ii < j);
```

Si hemos dejado de imprimir líneas porque no hay más valores en pila para continuar, hay un número de líneas que no existen y que romperían la ventana de pila, por lo que nos aseguraremos que si pasa, sean rellenadas con espacios.

```
        while (k < lim){
            printf("\033[%d;%dH\033[K|%*s|\n",
                H_REGS+k+2, w_code+1, w_stack-2, "");
            k++;
```

```
}
```

Por último, al terminar de imprimir la pila, pondremos el borde inferior de las ventanas de código y pila.

```
/* Code and Stack Box Bottom Line*/
printf("|-%*s-|-%*s-|\n", w_code-2, code, w_stack-2,
    stack);
```

Una vez finaliza el programa, `proc.c` llama a la función `exit_interface()`, que libera la memoria usada en este fichero.

```
void exit_interface(){
    free(lines);
    free(code);
    free(stack);
}
```

instr.c

El fichero `instr.c` es el más importante del programa, por detás de `proc.c`. Este fichero implementa todas las funciones que simulan el juego de instrucciones del i386, y las agrupa de manera que puedan ser llamadas fácilmente desde `proc.c`.

Inicializaciones En primer lugar, declaramos las variables globales que van a actuar como registros, y la que va a almacenar todo el espacio de direcciones de la memoria.

```
uint8_t * mem;
uint32_t eax = 0, edx = 0, esp = 0, esi = 0, eip = 0,
    cs = 0, ds = 0, fs = 0, ecx = 0, ebx = 0, ebp = 0,
    edi = 0, ss = 0, es = 0, gs = 0;
```

De manera adicional, son declarados otros registros secundarios que existen en el i386 real, pero como su uso está orientado a debug, y nosotros vamos a ejercer dicho debug de una manera distinta, no son utilizados. Otras variables que sí que vamos a utilizar deben ser redefinidas con la palabra reservada `extern`, ya que son declaradas en otros ficheros.

```
extern uint32_t eflags; // flags.c
extern int rows, cols;  // interface.c
extern arg_flags args;  // proc.c
```

A juego con los argumentos que obtengamos desde `proc.c`, debemos definir otras variables, que van a servir para llevar un registro en el propio `instr.c` de si dichas funciones han sido activadas o no.

```
uint8_t debug = 0, nx = 0;

/* File for debugging */
FILE * debugger;
```

Desde los argumentos también se define si las variables que definen el inicio y fin de la pila toman un valor por defecto, aleatorio, o el que indique el usuario, por lo que tenemos que definir variables globales que almacenen estos valores. Ya no podemos usar macros cuyos valores se decidían en tiempo de compilación, como se hacía en anteriores versiones.

```
uint32_t STACK_TOP, STACK_BOTTOM;
```

A continuación, definiremos dos vectores. El vector `void * regs []` contendrá punteros a registros, y el vector `uin8_t regs_size[]` contendrá el tamaño en bits de los registros. Estos vectores coinciden en sus índices. Es decir, si el índice 2 pertenece por ejemplo, al registro AL, en `regs[2]` obtendremos el puntero al registro AL y en `regs_size[2]` obtendremos 8, que es el número de bits del registro AL.

Estos vectores han sido declarados ya que la librería *Capstone*, a la hora de pasar argumentos, cuando un operando es un registro (`X86_OP_REG`), solo

incluye un número identificador, y es a partir de ese número identificador que debemos obtener el puntero a registro y su tamaño, aunque su tamaño también puede ser obtenido con el campo `size` del operando.

```
void * regs[] = {   NULL,                          // INVL
                (void *)(((uint8_t *)&eax)+1),     // AH
                &eax,                              // AL
                &eax,                              // AX
                (void *)(((uint8_t *)&ebx)+1),     // BH
                ...
                &eax,                              // EAX
                &ebp,                              // EBP
                &ebx,                              // EBX
                &ecx,                              // ECX
                ...
```

Como podemos ver, el primer índice de registro está reservado para indicar que el registro es inválido, y que cuando queremos indicar un puntero a uno de los dos registros de 8 bits dentro de un registro de 32 bits, debemos hacer lo siguiente.

- Para los 8 bits más bajos (AL, BL, CL...) basta con indicar el puntero a su registro de 32 bits, ya que al *castear* [1] el puntero `void *` a `uint8_t *`, el puntero apuntará únicamente al primer byte, y los otros 3 no serán modificados si se usa. Lo mismo pasa con los registros de 16 bits (AX, BX, CX...), ya que comparten el primer bit, pero estos deben ser convertidos a `uint16_t *`.
- Para los 8 bits siguientes (AH, BH, CH...) se debe convertir primero el puntero `uint32_t *` a `uint8_t *`, a continuación sumarle 1, para desplazar el puntero 8 bits hacia delante, y convertirlo a `void *` de nuevo.

El vector `regs_size` tiene la siguiente estructura.

```
/* REGISTER            INVL,   AH,   AL,   AX,   BH, ... */
/* INDEX                  0     1     2     3     4  ... */
uint8_t regs_size[] = {   0, 0x08, 0x08, 0x10, 0x08, ...}
```

[1] Conversión entre tipos compatibles en un lenguaje de programación.

También definimos el tipo `Instruction`, que corresponde a un puntero a función que toma un `struct * cs_insn` como argumento. Este tipo va a envolver todas y cada una de las instrucciones. Es decir, cada una de las funciones que implementan instrucciones encajan en esta definición, lo que nos va a permitir agruparlas en un vector.

```
typedef int(*Instruction)(cs_insn *insn);
```

A continuación, definimos los vectores que contienen tanto las cadenas con los nemotécnicos de las instrucciones, como los punteros a las funciones que implementan dichas instrucciones. Ambos vectores ya han sido explicados en el Apartado C.2.

```
const char *inss[] = {
"aaa","aad","aam","aas","adc","add","and", "arpl",
...
"rep outs", "rep stosb","rep stosw","rep stosd",""};

Instruction instructions[] = {
aaa_i, aad_i, aam_i, aas_i, adc_i, add_i, and_i, arpl_i,
...
rep_outs_i,rep_stos_i, rep_stos_i, rep_stos_i, NULL};
```

Funciones Auxiliares I Posteriormente, definimos las funciones auxiliares de suma y resta que permiten realizar operaciones seguras de lectura y escritura sobre la pila, sin cruzar los límites establecidos por las variables `STACK_TOP` y `STACK_BOTTOM`, tal como se indica en el Apartado C.2.

```
void esp_minus(uint8_t v){
    esp -= (esp - v < STACK_TOP)? 0: v;
}

void esp_plus(uint8_t v){
    esp += (esp + v > STACK_BOTTOM)? 0 : v;
}
```

Función `initialize()` En lo que al flujo del programa completo respecta, la primera función llamada por `proc.c` que corresponde al fichero `instr.c` es `int initialize()`, que se encarga de inicializar las estructuras de datos del simulador, para que se encuentren en una forma consistente cuando `proc.c` vaya a llamar al *loader* para cargar el programa. En esta función es donde se comprueban los argumentos introducidos en `proc.c` para ajustar el comportamiento a lo que el usuario ha pedido. Por ejemplo, la primera comprobación que se realiza es si el ASLR está activo. Se confía en que no hay argumentos incompatibles ya que esa comprobación se ha hecho en el fichero principal, en la función `main()`.

```
/* ASLR (only stack because code is no-pie) */
if ((args.flags & ARGS_ASLR_FLAG) >> ASLR_FLAG){
    uint8_t num = random() % 16;
    STACK_BOTTOM = 0xffff0000 | (num << 12);
    STACK_TOP = STACK_BOTTOM - 0x22000;
    esp = STACK_BOTTOM;
}
```

En la comprobación de si el ASLR ha sido indicado como opción, si ésta es afirmativa, primero generamos un número aleatorio entre 0x0 y 0xF. Este número lo desplazamos 12 posiciones hacia la izquierda, para que resida del bit 12 al 15, y aplicaremos una operación OR con `0xFFFF0000`, para que, si por ejemplo el número aleatorio resultante es 0xB, la dirección resultante sea `0xFFFFB000`, que se indica como *Stack Pointer* inicial y `STACK_BOTTOM`, y tras restarle `0x22000`, obtenemos el otro límite de pila, `STACK_TOP`.

Esto nos otorga una dirección inicial de pila, aunque con sólo 4 bits de entropía. Por lo que si un usuario probara un ataque que implicara direcciones específicas de pila[2], tendría éxito una de cada dieciséis veces. Obviamente no es una implementación infalible pero sirve como ejemplo de como funciona la randomización del espacio de direcciones.

Si por el contrario, el ASLR no está indicado como opción, pasamos a comprobar si el usuario ha especificado valores manuales para `STACK_BOTTOM` y `STACK_TOP`. Si tampoco ha indicado valores manuales, pasamos a los valores por defecto.

[2]Un ataque que tuviera como requisito saber en qué dirección de memoria comienza la pila, ya que con ella se puede calcular la dirección de un buffer extrapolando otras ejecuciones, por ejemplo.

```
/* Custom stack range*/
else if((args.stack_top != 0)
        && (args.stack_bottom != 0)){
    STACK_BOTTOM = args.stack_bottom;
    STACK_TOP = args.stack_top;
    esp = STACK_BOTTOM;
}else{
    STACK_BOTTOM = 0xFFFFA000;
    STACK_TOP = STACK_BOTTOM - 0x22000;
    esp = STACK_BOTTOM;
}
```

A continuación, comprobamos si se ha indicado la opción NX, para evitar la ejecución de instrucciones en pila, y si es así, activamos la variable que almacena esta opción. Esta variable será comprobada más adelante, en la función `dispatcher()`, y comprobará si el registro EIP contiene una dirección que reside en pila o no.

```
/* NX - (Non-Executable Stack) */
if ((args.flags & ARGS_NX_FLAG) >> NX_FLAG){
    nx = 1;
}
```

Por otro lado, también comprobamos si se ha indicado la opción de *debug*, y si es así, abrimos el arhcivo de debug y almacenamos tanto la flag de depuración, como el puntero a archivo. Una vez abierto el fichero, escribiremos el rango actual de pila. Además, cada vez que se ejecute la función `dispatcher()`, si la variable `debug` está a 1, se escribirá el valor actual de los registros. Esta funcionalidad cobra más sentido y está explicada con detalle en el Anexo G.1.

```
if ((args.flags & ARGS_DEBUG_FLAG) >> DEBUG_FLAG){
    debug = 1;
    debugger = fopen("debugger.log", "w");
    if (!debugger){
        debug = 0;
    }else{
        fprintf(debugger, "STACK RANGE: %08x-%08x\n",
```

```
                STACK_TOP, STACK_BOTTOM);
        }
    }
```

Cuando ya hemos procesado los argumentos del programa, pasamos a inicializar los registros e inicializar la memoria. Todos los registros están inicializados a cero, menos el registro EFLAGS que tiene asignado su valor por defecto. Por ello, solo tendremos que inicializar los selectores de segmento, y la memoria. El macro `UINT32_MAX` en tiempo de compilación se reemplaza por el valor 2^{32}. La memoria se reserva mediante la función `calloc` para que los bytes de todas las direcciones de memoria estén inicializados a `0x00`.

```
/* Initialize segments */
cs = 0x23;
ds = 0x2b;
ss = 0x2b;
es = 0x2b;
fs = 0x00;
gs = 0x00;

/* Allocate 4GB of memory */
mem = (uint8_t *)calloc(sizeof(uint8_t), UINT32_MAX );
if (!mem)
    return 0;
```

Por último, inicializamos la GDT, que ha sido explicada en el Apartado C.2. A pesar de tener una segmentación *Flat*, a la hora de resolver una dirección de memoria efectiva, se siguen usando los selectores de segmentos y segmentos, solo que tienen la base puesta a 0 y el límite al valor máximo, por lo que obtenemos la misma dirección de memoria que obtendríamos sin segmentación. A la hora de inicializar la GDT, usamos un macro definido en `instr.h`, que establece una dirección de memoria en el espacio del kernel para almacenar la GDT, y a continuación guardamos dicha dirección de memoria en el registro GDTR, así como donde acaba la GDT.

```
/* Initialize GDT at GDT_ADDR (defined in instr.h) */
GDT_Descriptor * gdt = (GDT_Descriptor *)(mem + GDT_ADDR);
init_gdt(gdt);
```

```
gdtr.base = GDT_ADDR;
gdtr.limit = GDT_ENTRIES * sizeof(GDT_Descriptor);
```

La función `init_gdt()` crea tantas entradas como indique el macro `GDT_ENTRIES`, declarado en `instr.h`, e inicializa varios campos en función de para qué vayan a estar destinados dichos segmentos. Este es un proceso demasiado complejo y sin demasiada influencia en el código. Para más información, es conveniente revisar el Manual de Referencia del i386 [4].

Por último, la función `initialize()` devuelve 1, indicando que la inicialización ha sido satisfactoria. Esto viola el estándar de facto, en el que las funciones y programas devuelven 0 si todo ha salido bien, y 1 o el código de error si algo ha salido mal. En este caso, se ha decidido hacerlo de esta manera para favorecer la comprensión del código, ya que en la llamada a esta función, se incluye dentro de una sentencia if que comprueba si la inicialización ha sido correcta.

```
/* Initializes some registers and
    allocates memory for mem variable */
if(!initialize())
   return 1;
```

De esta manera, el predicado lógico a comprobar si la inicialización del programa ha fallado, y por tanto salir del programa con código de error, es `!initialize()`, que implícitamente expresa "Si la inicialización falla".

Si lo hiciéramos según el estándar de facto, el predicado en cuestión sería `if(initialize())`, que es mucho más contraintuitivo para un programador ajeno que lea el código fuente.

Funciones auxiliares II Antes de entrar en materia de las implementaciones de las instrucciones, debemos detallar una serie de otras funciones auxiliares que son utilizadas en gran parte de las instrucciones. Podemos ver su signatura en `instr.h`, donde están separadas de las funciones de las instrucciones.

```
uint32_t reg_val(int reg_id);
uint32_t eff_addr(x86_op_mem m);
```

```
uint32_t pow_i(uint32_t b, uint32_t exp);
void write16(uint32_t addr, uint16_t value);
void write32(uint32_t addr, uint32_t value);
uint16_t read16(uint32_t addr);
uint32_t read32(uint32_t addr);
uint32_t sign_extend8_32(uint8_t v);
uint32_t sign_extend16_32(uint16_t v);
uint16_t sign_extend8_16(uint8_t v);
uint64_t sign_extend32_64(uint32_t v);
uint32_t zero_extend16_32(uint16_t v);
uint32_t zero_extend8_32(uint8_t v);
uint16_t zero_extend8_16(uint8_t v);
```

Las funciones que contienen `zero_extendA_B` o `sign_extendA_B` son triviales, ya que únicamente devuelven el valor de tamaño B, tras hacer una extensión de ceros (o de singo) hasta el tamaño B del valor de tamaño A. La extensión de ceros se traduce como un *casteo* al uso, y la extensión de signo se traduce como testear el bit más significativo de un valor de tamaño A, y añadir tantos bits de igual valor a ese bit, hasta que el tamaño sea igual al tamaño B. El siguiente fragmento de código es un ejemplo de esta mecánica.

```
uint32_t sign_extend8_32(uint8_t v){
    return (v & 0x80)?(v | 0xFFFFFF00):v;
}
```

Si el bit 7 está activado, se le aplica un `OR 0xFFFFFF00` para añadir 24 unos delante (para convertirlo en un número negativo), y si no, se devuelve el valor tal cual, lo que implícitamente llama a un *casteo* a `uint32_t`, que es el tipo de retorno de la función, y equivale a una extensión de ceros.

Por ello, las funciones realmente importante son el resto de funciones. En primer lugar, las funciones `read16` y `read32` permiten leer un valor en formato *Little Endian* de una dirección de memoria.

```
/**
 * Returns a word (16 bits) from the address given.
 *
 * @param addr addres to read from
```

```
 */
uint16_t read16(uint32_t addr) {
    return mem[addr] | (mem[addr + 1] << 8);
}

/**
 * Returns a doubleword (32 bits) from the address given.
 *
 * @param addr addres to read from
 */
uint32_t read32(uint32_t addr) {
    return mem[addr] |
           (mem[addr + 1] << 8) |
           (mem[addr + 2] << 16) |
           (mem[addr + 3] << 24);
}
```

Para facilitar la comprensión del código, se han planteado como tantas lecturas según bytes tenga el valor a leer, y luego desplazamiento de dichos bytes en función de donde correspondan, pero podría haberse planteado con un *casteo* de puntero, ya que `mem` es un puntero `uint8_t *`, por lo que al desreferenciarlo, solo otorga un byte.

Con el método de casteo primero obtendríamos el puntero a dicha dirección de memoria simulada, y castearíamos dicho puntero al tipo que correspondiera, para que al desreferenciarlo, devuelva el tipo que queremos.

```
uint32_t read32(uint32_t addr){
    return *((uint32_t *)(mem+addr));
}

uint16_t read16(uint32_t addr){
    return *((uint16_t *)(mem+addr));
}
```

Por otro lado, las funciones `write16` y `verb32` permiten la función inversa, escribir un valor en *Little Endian* en una dirección de memoria.

```
/**
 * Writes a word (16 bits) into the address given.
 *
 * @param addr addres to write on
 * @param value to write
 */
void write16(uint32_t addr, uint16_t value) {
    mem[addr]     = value & 0xFF;
    mem[addr + 1] = (value >> 8) & 0xFF;
}

/**
 * Writes a doubleword (32 bits) into the address given.
 *
 * @param addr addres to write on
 * @param value to write
 */
void write32(uint32_t addr, uint32_t value) {
    mem[addr]     = value & 0xFF;
    mem[addr + 1] = (value >> 8) & 0xFF;
    mem[addr + 2] = (value >> 16) & 0xFF;
    mem[addr + 3] = (value >> 24) & 0xFF;
}
```

De nuevo, se han planteado estas operaciones con el paradigma de desplazamiento para facilitar la comprensión, aunque con casteo de punteros podrían haberse planteado de la siguiente manera.

```
void write32(uint32_t addr, uint32_t value){
    *((uint32_t *)(mem+addr)) = value;
}

void write16(uint32_t addr, uint16_t value){
    *((uint16_t *)(mem+addr)) = value;
}
```

Ninguna de estas implementaciones es mejor ni peor. La implementación basada en el casteo de punteros es más compacta, pero podría no funcionar

si se ejecuta bajo unas condiciones específicas. Por lo tanto, la forma menos compacta nos asegura portabilidad, sobre todo en arquitecturas que podrían utilizar *Big Endian*.

Otra función esencial es `eff_addr()`. Esta función permite obtener la dirección de memoria efectiva a partir de un operando de memoria (`x86_op_mem`). Esto se debe a que las direcciones de memoria no siempre son formadas con un valor inmediato, o con el contenido de un registro, sino que a estos valores se les puede aplicar desplazamiento relativo o absoluto, por lo que resulta especialmente útil una función que obtenga el resultado final para poder operar con la dirección de memoria, sea como sea el operando.

```
/**
 *  Obtains an effective address
 *       using a x86_op_mem argument.
 *
 * @param m struct containing all addressing variables.
 *
 * @return effective address.
 */
uint32_t eff_addr(x86_op_mem m){
    uint32_t base = 0, index = 0, disp = 0, segment = 0;
    uint32_t scale = 1;
    if (m.base != X86_REG_INVALID)
        base = reg_val(m.base);
    if (m.index != X86_REG_INVALID)
        index = reg_val(m.index);
    if (m.scale != 0)
        scale = m.scale;
    disp = m.disp;
    if (m.segment != X86_REG_INVALID){
        segment = reg_val(m.segment);
        segment = get_gdt_base((uint16_t)segment);
    }

    return (uint32_t)((int32_t)segment + (int32_t)base +
        (int32_t)index*(int32_t)scale + (int32_t)disp);

}
```

Como podemos ver, podemos tener los siguientes valores a la hora de calcular una dirección de memoria.

- Base : es el valor de un registro. Funciona como base a la que se le van sumando el resto de valores.
- Segmento : es el segmento obtenido a partir de un selector de segmento que puede estar presente. Como hemos simulado una arquitectura *Flat*, la base siempre será 0 y el límite será el máximo, por lo que su comprobación no es necesaria. En una segmentación convencional, si la dirección efectiva final supera el límite del segmento, se produce una excepción por fallo de segmentación.
- Escala (scale) : valor que multiplica el índice para ofrecer mayor desplazamiento.
- Index : valor que es multiplicado por la escala. Es otro registro.
- Desplazamiento : valor adicional que se suma a la operación para conseguir una mayor maniobrabilidad.

En la función podemos ver que todos estos valores se inicializan a cero, y se va comprobando si el operando los contiene o por el contrario son nulos. Al final de la función se realiza la operación estándar, ya que si un valor no está presente, simplemente ha sido inicializado a 0 y se anula. La escala es el único valor que se inicializa a 1, por si éste es nulo pero el index no, para que no lo multiplique por cero y lo anule.

Con esto, obtenemos una dirección de memoria efectiva. Por ejemplo, con `EAX = 0x100` y `ECX = 0x2`, la expresión `[EAX + ECX*4 + 0x10]`, el registro EAX sería la base, ECX el índice, 4 la escala, y `0x10` el desplazamiento. Por lo tanto la dirección de memoria efectiva sería `0x118`, y si la instrucción fuera MOV, por ejemplo, se movería el valor apuntado por la dirección de memoria `0x118`.

La otra función esencial, justo la acabamos de ver en la función anterior, y es `reg_val()`. Esta función devuelve el valor de un registro en función del índice del registro, según lo que hemos podido ver en la inicialización de los vectores de registros, en el Apartado D.2. Por ejemplo, si de una instrucción extraemos un operando, y ese operando es un registro, siempre va de la mano con un número de registro.

A partir de el *struct* de la instrucción, `cs_insn insn`, obtenemos los detalles de esa instrucción (`insn->detail->x86`). Con los detalles de la

instrucción obtenemos el operando. Los operandos tienen un campo `type`, que si equivale al macro `X86_OP_REG`, significa que el campo `reg` del operando no será nulo, y habrá en él el identificador de un registro.

```
cs_x86 x86 = insn->detail->x86;
cs_x86_op op1 = x86.operands[0];
cs_x86_op op2 = x86.operands[1];

uint8_t s1 = op1.size, s2 = op2.size;

uint32_t val;
if (op2.type == X86_OP_REG){
    val = reg_val(op2.reg);
```

Es este identificador de registro el que le pasamos a la función `reg_val()`. La función accede a la posición del vector `regs` que coincida con el identificador, y obtiene el valor en función del tamaño en bits del registro, al que aplica una extensión de ceros hasta 32 bits, ya que es el máximo tamaño que puede tomar un valor en el i386.

```
/**
 * Obtains a register value using a reg_id.
   Reg_id most likely comes from a operand
   where type is x86_op_reg.
 */
uint32_t reg_val(int reg_id){
    if (reg_id < 0 || reg_id > 49){
        return -1;
    }
    void * p = regs[reg_id];
    uint8_t base = regs_size[reg_id];
    if (p != NULL && base != 0){
        switch(base){
            case 0x8:
                return (uint32_t)*((uint8_t *) p);
                break;
            case 0x10:
                return (uint32_t)*((uint16_t *) p);
                break;
```

```
                case 0x20:
                    return (uint32_t)*((uint32_t *) p);
                    break;
                default:
                    return (uint32_t)*((uint32_t *) p);
                    break;
            }
        }

        return 0;
    }
```

Funciones que implementan instrucciones En cuanto a las instrucciones, las funciones que las implementan siguen la misma signatura, tal como se ha mencionado en el Apartado D.2. Sin embargo, también siguen la misma estructura. Esta estructura es la norma general, aunque ciertas instrucciones específicas (como aquellas con `REP`, `RET`, `CALL`) pueden diferir.

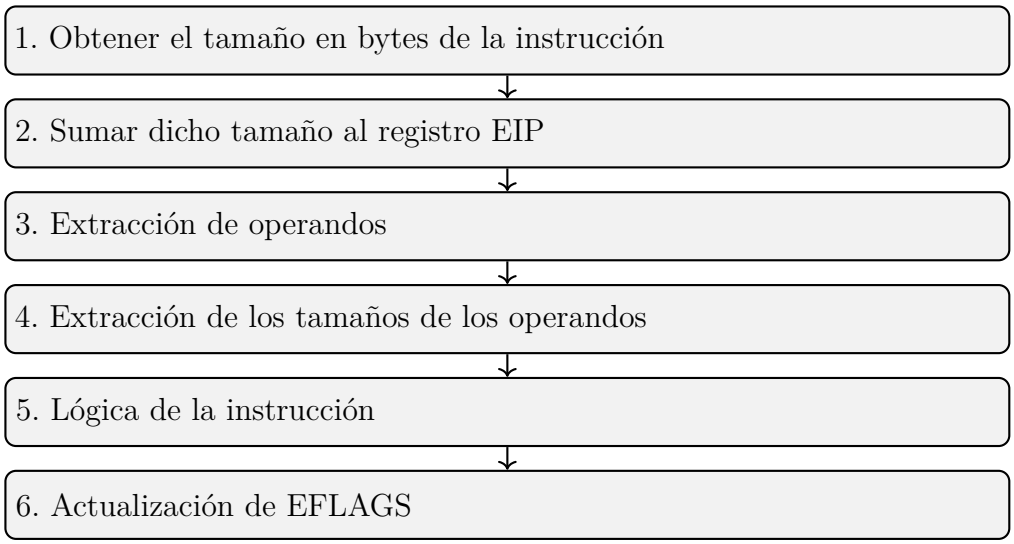

Figura D.3: Proceso general de simulación de una instrucción

Todas las funciones que implementan instrucciones tienen en su nombre el nemotécnico de la instrucción en minúscula, seguido de un "_i". Con esto indicamos que son instrucciones y que siguen la signatura del tipo `(*Instruction)(cs_insn *insn)`. Con el fin de documentar de manera

adicional las instrucciones, cada función va precedida de un comentario multilínea que incluye el nemotécnico de la instrucción, una breve explicación, los *opcodes* a los que puede corresponder, las excepciones que provoca y el efecto que produce en las banderas.

```
/**
 *  POP. Pop a word from the stack.
 *
 *  Opcodes 0x8F /0, 0x58 + r, 0x1F, 0x07, 0x17,
 *   0x0F A1, 0x0F A9.
 *
 *  Segment and Page Exceptions in Protected Mode.
 *
 *  No flags affected.
 *
 *  @param insn instruction struct that
 *          stores all the information.
 */
int pop_i(cs_insn *insn){
    eip += insn->size;
    cs_x86 x86 = insn->detail->x86;
    cs_x86_op op1 = x86.operands[0];
    uint8_t s = op1.size;

    if (op1.type == X86_OP_REG){
        void * p = regs[op1.reg];
        if (s == 2){
            *((uint16_t *)p)=read16(esp);
            esp_plus(2);
        }else{
            *((uint32_t *)p)=read32(esp);
            esp_plus(4);
        }
    }else if(op1.type == X86_OP_MEM){
        uint32_t addr = eff_addr(op1.mem);
        write32(addr, read32(esp));
        esp_plus(4);
    }
    return 0;
}
```

proc.c

El fichero `proc.c` es el más importante de la rama de Linux, ya que se encarga de interactuar con el usuario, recoger las opciones que ha seleccionado, y delegar esas operaciones en el resto de ficheros. A continuación vamos a proceder a explicar el funcionamiento del fichero proc.c.

Procesamiento de argumentos En primer lugar, debemos comprender los argumentos que pueden ser indicadas por el usuario, de cara a la ejecución del programa. Para ello, deberemos prestar atención a los macros definidos en el fichero de cabecera `proc.h`, ya que van a definir cada una de las opciones posibles.

```
#define DEBUG_FLAG     0
#define NX_FLAG        1
#define ASLR_FLAG      2
#define INTERFACE_FLAG 3
#define NORMAL_FLAG    4
#define RAW_FLAG       5
#define ELF_FLAG       6
```

Con esto, indicamos un número para cada opción, que se va a convertir en un bit de una variable de *flags*. En dicha variable de *flags*, si cogemos como ejemplo el macro `NX_FLAG` que corresponde al número 1, el bit 1 estará a 1 si el usuario ha indicado esa opción, o estará a 0 si el usuario no la ha indicado.

Por ello, una vez tenemos los números de los bits, podemos obtener números binarios que actúen como máscara para comprobar dichos números. La operación `<<` indica un desplazamiento de bits hacia la izquierda, por lo que para cada opción, estamos obteniendo un número binario al desplazar un 1 tantas veces a la izquierda como el número de la propia bandera.

```
#define ARGS_DEBUG_FLAG        (1 << DEBUG_FLAG)
#define ARGS_NX_FLAG           (1 << NX_FLAG)
#define ARGS_ASLR_FLAG         (1 << ASLR_FLAG)
#define ARGS_INTERFACE_FLAG    (1 << INTERFACE_FLAG)
#define ARGS_NORMAL_FLAG       (1 << NORMAL_FLAG)
#define ARGS_RAW_FLAG          (1 << RAW_FLAG)
#define ARGS_ELF_FLAG          (1 << ELF_FLAG)
```

Podemos verlo de una manera más estructurada para comprenderlo mejor.

Argumento del programa	Número	Máscara
DEBUG	0	0b00000001
NX	1	0b00000010
ASLR	2	0b00000100
INTERFACE	3	0b00001000
NORMAL	4	0b00010000
RAW	5	0b00100000
ELF	6	0b01000000

Tabla D.4: Números y máscaras de las opciones iniciales del simulador

La última definición que se realiza en el fichero `proc.h` es el `struct` que se encarga del almacenamiento de las opciones.

```
typedef struct {
   uint8_t flags;
   uint32_t stack_top;
   uint32_t stack_bottom;
   uint8_t optind;
}arg_flags;
```

Podemos ver que el `struct arg_flags` tiene 4 campos de distintos tamaños.

- `uint8_t flags` es una variable de 8 bits que funciona como registro de banderas. No hacen falta más bits ya que solo tenemos 7 opciones a indicar, que se traducen a 7 bits de banderas. Esta es la variable cuyo bit 1 estará activo si el argumento `--nx (-x)` ha sido pasado al programa.
- `uint32_t stack_top` Corresponde con el argumento `--stack-top (-t)`, que sirve para reemplazar el límite bajo de la pila por defecto. Si dicha opción se indica, el valor obligatorio que debe acompañarla se almacena aquí.
- `uint32_t stack_bottom` Corresponde con el argumento `--stack-bottom (-b)`, que sirve para reemplazar el límite alto de la pila por defecto. Si dicha

opción se indica, el valor obligatorio que debe acompañarla se almacena aquí.

- `uint8_t optind` Corresponde al índice de argumentos que dejan de corresponder a argumentos del simulador y comienzan a ser el nombre del fichero a ejecutar y sus argumentos. Por ejemplo, si indicamos 3 opciones, digamos `./proc -i -e -d programa opcion1`, la variable `optind` equivaldrá a 4, ya que es `argv[4]` el primer argumento que no es una opción que indicar al programa, sino que es el programa a ejecutar por el simulador.

Vamos a emplear este tipo estructurado y lo vamos a definir como una variable global, junto con algunas variables auxiliares para manipular los argumentos.

```
/* Program args */
arg_flags args;

/* Stack range limits [declared on instr.c] */
extern uint32_t STACK_BOTTOM, STACK_TOP;

/* File pointer to debug and debug flag [declared on instr.c] */
extern FILE * debugger;
extern uint8_t debug;
```

A continuación, vamos a ver como se gestionan estos argumentos desde la función principal. Para ello, deberemos definir un struct con las opciones posibles. Pero primero deberemos incluir la cabecera `getopt.h`.

```
#include <getopt.h>
```

Esta cabecera nos otorga el tipo estructurado que vamos a emplear a continuación.

```
int main(int argc, char *argv[], char *envp[]){

   /* Define supported options */
   static struct option long_options[] = {
        {"debug", no_argument, 0, 'd'},
```

```
        {"nx", no_argument, 0, 'x'},
        {"stack-top", required_argument, 0, 't'},
        {"stack-bottom", required_argument, 0, 'b'},
        {"aslr", no_argument, 0, 'a'},
        {"help", no_argument, 0, 'h'},
        {"interface", no_argument, 0, 'i'},
        {"normal", no_argument, 0, 'n'},
        {"elf-mode", no_argument, 0, 'e'},
        {"raw-mode", no_argument, 0, 'r'},
      {0,0,0,0}
    };
```

Posteriormente inicializaremos los campos de la variable global `arg_flags args`.

```
    /* Init args object */
    args.flags = 0;
    args.stack_top = 0;
    args.stack_bottom = 0;
    args.optind = 0;
```

A continuación, iteraremos sobre los argumentos, y o bien los guardaremos, o si corresponden a la ayuda, la mostraremos y saldremos del programa.

```
     int opt;
  while ((opt = getopt_long(argc, argv, "dxt:b:aienrh",
                  long_options, NULL)) != -1) {
     switch (opt) {
        case 'h':
           print_help();
           return 0;
        case  'd': args.flags |= ARGS_DEBUG_FLAG; break;
        case  'x': args.flags |= ARGS_NX_FLAG; break;
        case  'a': args.flags |= ARGS_ASLR_FLAG; break;
        case  'i': args.flags |= ARGS_INTERFACE_FLAG; break;
        case  'n': args.flags |= ARGS_NORMAL_FLAG; break;
        case  'r': args.flags |= ARGS_RAW_FLAG; break;
        case  'e': args.flags |= ARGS_ELF_FLAG; break;
   case  't': args.stack_top = strtoul(optarg, NULL, 0);break;
   case  'b': args.stack_bottom = strtoul(optarg, NULL, 0);break;
```

```
       case '?':
        print_help();
          return 1;
    }
 }
   /* Store index where end options and begin the arguments*/
 args.optind = optind;
```

La función `print_help()` únicamente imprime por pantalla cómo usar los argumentos del programa, por lo que al ser trivial, la ignoraremos.

Una vez hemos extraído los argumentos indicados por el usuario, deberemos comprobar que el usuario no haya seleccionado argumentos incompatibles. Para lo que utilizaremos la función `validate_opts()`.

```
    /* Validate options */
   int val_opts = validate_opts(args, argc);
   if (val_opts){
      return 1;
   }
```

Dicha función esta definida antes de la función `main` y comprueba incompatibilidades en los argumentos.

```
int validate_opts(arg_flags args, int argc){
   /* Mutually exclusive INTERFACE and NORMAL */
   if (((args.flags & ARGS_INTERFACE_FLAG) >> INTERFACE_FLAG)
        && ((args.flags & ARGS_NORMAL_FLAG) >> NORMAL_FLAG )){
      printf(" Bad options, -i (--interface) and -n (--normal)
               cannot be selected at the sime time\n\n");
      goto bad_args;
   }

   ...

   if(!((args.flags & ARGS_RAW_FLAG) >> RAW_FLAG)
        && !((args.flags & ARGS_ELF_FLAG) >> ELF_FLAG)){
      printf(" Bad options, one of the following must be
               present. -e, -r\n\n");
      goto bad_args;
```

```
   }

   if(args.optind >= argc){
      printf(" Bad options. A 32-bit program should be
               specified so it can be simulated\n\n");
      goto bad_args;
   }

   return 0;

   /* Tag that allows different cases to jump to same result */
   bad_args:

   print_help();
   return 1;
}
```

Una vez procesados los argumentos, deberemos modificar los argumentos que le vamos a pasar.

```
    /* Move args pointer to 32bit-program specific args */
   argc -= optind;
   argv += optind;
```

De esta manera, en `argc` tendremos el número de argumentos que son únicamente del programa simulado, y en `argv` únicamente los punteros a los argumentos del programa simulado.

Por último, comprobamos las 4 opciones acerca de como ejecutar el simulador, y en función de la que sea (blind o interface, raw o elf), ejecutamos una función u otra.

```
    /* Args should be OK if program reaches this line */
   if ((args.flags & ARGS_INTERFACE_FLAG) >> INTERFACE_FLAG){
      if ((args.flags & ARGS_RAW_FLAG) >> RAW_FLAG){
         return interface_raw_main(argc, argv, envp);
      }else if((args.flags & ARGS_ELF_FLAG) >> ELF_FLAG) {
         return interface_elf_main(argc, argv, envp);
      }
   }else if(((args.flags & ARGS_NORMAL_FLAG) >> NORMAL_FLAG )){
```

```
      if ((args.flags & ARGS_RAW_FLAG) >> RAW_FLAG){
         return blind_raw_main(argc, argv, envp);
      }else if((args.flags & ARGS_ELF_FLAG) >> ELF_FLAG){
         return blind_elf_main(argc, argv, envp);
      }
   }else{
      print_help();
      return 1;
   }
```

Inicializaciones Además de las variables globales que hemos mencionado anteriormente, en el fichero `proc.c` se declaran (o referencian) otras variables globales.

```
#define MIN(a,b) ((a > b)? (b) : (a))
#define MAX_STR 75
#define MAX_BRK 25

/* MEMORY [declared on instr.c] */
extern uint8_t * mem;
/* REGISTERS [declared on instr.c] */
extern uint32_t eflags;
extern uint32_t eax, edx, esp, esi, eip, cs, ds, fs, ecx, ebx,
    ebp, edi, ss, es, gs;

/* Screen display size [declared on interface.c] */
extern int rows, cols;
```

Ya conocemos las variables de los registros y la memoria, así como las filas y columnas de la interfaz. Sin embargo, disponemos de un macro `MIN(a,b)` que actúa como función para conseguir el valor menor de entre dos valores. El macro `MAX_STR` indica el máximo de bytes que pueden ocupar las cadenas en algunos contextos, a la hora de reservar espacio, mientras que el macro `MAX_BRK` establece el número máximo de *breakpoints* que puede establecer un usuario en el modo interfaz.

Funciones auxiliares Antes de entrar en materia de las funciones que implementan los 4 modos de ejecutar el programa simulado, vamos a detallar una serie de funciones auxiliares, que son usadas en algunas de estas funciones principales. En primer lugar, tenemos la función auxiliar `contains()`, que

se utiliza para comprobar si un valor está contenido en un array de un determinado tamaño. Es usado para comprobar si la dirección de memoria actual está en la tabla de *breakpoints* cuando ejecutamos una instrucción.

```
/**
 *  Returns 1 if a value is contained in
 *      a certain array or 0 if not.
 *
 *  @param arr is the array to check in.
 *  @param size is arr's size
 *  @param val is the value to look for.
 *
 *  @return 1 if val is found or 0 if not.
 */
uint8_t contains(uint32_t arr[], size_t size, uint32_t val){
   for (int i=0; i<size; i++){
      if (arr[i] == val)
         return 1;
   }
   return 0;
}
```

A continuación vamos a explicar cada una de las cuatro funciones principales. Cada una ejecuta un fichero distinto de una manera distinta, y todas tienen la misma signatura. Toman como argumentos el valor `argc` (una vez restado optind), y los arrays `argv` y `envp`.

- `blind_raw_main(argc, argv, envp);`
- `blind_elf_main(argc, argv, envp);`
- `interface_raw_main(argc, argv, envp);`
- `interface_elf_main(argc, argv, envp);`

En cada una de estas funciones se utiliza la función `cs_disasm()`, que toma los siguientes argumentos.

- **Handler** : el primer argumento es el handler que incluye las directrices para desensamblar, como la arquitectura o si incluir detalles o no. Aquí incluiremos nuestra varaible `csh handle`.

- **Buffer** : el segundo argumento es el buffer que contiene el bytecode, por lo que deberemos pasarle la dirección de memoria apuntada por el registro EIP si vamos a desensamblar una sola instrucción, o el inicio de las instrucciones si vamos a desensamblar todas. La sintaxis es `&mem[addr]`.
- **Límite** : el tercer argumento es el límite de bytes a desensablar, que cobra especial imporancia si vamos a desensamblar todas las instrucciones y no solo una. Debemos indicarle la última dirección de memoria que contiene instrucciones.
- **Dirección de memoria** : el cuarto argumento corresponde a la dirección de memoria de la primera instrucción que se va a desensamblar, ya que se le asignará dicha dirección de memoria y las de las siguientes instrucciones e calcularán de manera relativa a esa.
- **Número de instrucciones a desensamblar** : el quinto argumento corresponde al número de instrucciones a desensamblar. Si indicamos un 1 se desensamblará una sola instrucción, pero si indicamos un 0, se desensamblarán tantas como haya hasta llegar al límite impuesto por el tercer argumento.
- **Puntero** : el sexto argumento corresponde al puntero donde se van a almacenar los `structs` de las instrucciones que hayan sido desensambladas. Es la propia función la que se encarga de reservar memoria para ellas, así que basta con definir un `cs_insn *` y pasárselo a la función.

Por ello, si quisiéramos desensamblar una sola instrucción (la apuntada por el EIP) podríamos ejecutar lo siguiente, y tendríamos el resultado en `ins[0]`

```
cs_insn * ins;
cs_disasm(handle, &mem[eip], 16, eip, 1, &ins);
```

Si quisiéramos desensamblar tantas como tenemos en memoria, por ejemplo de `0x08048000` hasta `0x08049000`, haríamos lo siguiente. En `insn` tendríamos un array con los structs de las instrucciones, y en la variable `disasm` tendríamos el número total de instrucciones desensambladas.

```
cs_insn * insn;
```

```
size_t disasm = cs_disasm(handle, &mem[0x08048000],
    0x080489000 - 0x08048000, 0x08048000, 0, &insn);
```

Cuando expliquemos la función `interface_elf_main` veremos que será necesario un `cs_insn **` en vez de un `cs_insn *` ya que las secciones ejecutables no siempre son continuas, por lo que podría haber bytes indefinidos entre medias que contaminaran el desensamblaje, por lo que deben ser desensambladas sección a sección.

```
cs_insn ** insns;
...
cs_disasm(handle, &mem[addr], siz, addr, 0, &insns[i]);
```

Blind_raw_main Corresponde a la función que implementa la simulación de un programa *Raw*, es decir, binario sin cabeceras, y sin interfaz, ejecutando el programa sin mostrar nada por la terminal, excepto los valores que pueda imprimir el propio programa simulado.

En primer lugar, inicializamos el simulador, lo que inicializa la memoria y el resto de estructuras de datos del simulador. A continuación leemos el fichero binario y lo cargamos en memoria, obteniendo en la variable `size` el número de bytes cargados en memoria. Por último, definimos la variable `res` (resultado), que contendrá el valor temporal devuelto por cada instrucción ejecutada, para comprobar si han devuelto un valor que indique que debemos detener la ejecución.

```
int blind_raw_main(int argc, char *argv[], char *envp[]){

   /* Initializes some registers and allocates
            memory for mem variable */
   if(!initialize())
      return 1;

   uint32_t size;

   if(read_raw_file(argc, argv, envp, &size)){
      perror("Error reading raw file");
      goto exit;
   }

   uint32_t res;
```

A continuación, procedemos al desensamblaje, mediante las funciones que provee la librería *Capstone*. En primer lugar declaramos un handler (`csh handle`) y una instrucción `cs_insn *ins`. A continuación indicamos que el handler queremos establecerlo para arquitectura X86 de 32 bits, y que al desensamblar, queremos la opción de detalles encendida, ya que es la que nos otorga los operandos detallados que necesitamos para simular las instrucciones.

```
/* Handler for disassembling */
csh handle;
/* ins is the step by step instruction to execute */
cs_insn *ins;

/* Sets arch */
if (cs_open(CS_ARCH_X86, CS_MODE_32, &handle) != CS_ERR_OK){
   perror("setarch");
   return 1;
}

/* Activate detail feature. Useful for
     obtaining operands and other info */
cs_option(handle, CS_OPT_DETAIL, CS_OPT_ON);
```

Por último, ejecutamos el bucle principal de ejecución de instrucciones. Si falla el desensamblaje de una instrucción, o si una instrucción era de tipo INT y ha devuelto el valor `0xdeadbeef`, que implica haber ejecutado `do_exit()`, saltamos a la etiqueta exit, que libera la memoria y finaliza el programa. Si no falla nada, ni hemos ejecutado `do_exit()`, seguimos ejecutando instrucciones mediante las llamadas a `dispatcher()`.

```
while (true){

   /* Disassemble the current instruction.
         Bad alignment could have happened. */
   if(!cs_disasm(handle, &mem[eip], size-eip, eip, 1, &ins))
      // If number of disasm instructions is 0.
      goto exit;

   res = dispatcher(ins[0].mnemonic, &ins[0]);
   if (ins[0].bytes[0] == 0xCD && res == 0xdeadbeef){
```

```
        goto exit;
    }

    cs_free(ins, 1);
    ins = NULL;
}

exit:
if (debugger){
    fclose(debugger);
}
free(mem);
return 0;
```

Blind_elf_main Corresponde a la función que implementa la simulación de un programa ELF y sin interfaz, ejecutando el programa sin mostrar nada por la terminal, excepto los valores que pueda imprimir el propio programa simulado. Es exactamente igual a `blind_raw_main` excepto en dos puntos muy específicos.

En `blind_raw_main` era implícito que la primera dirección de memoria era 0, y la última era la que correspondía a la variable `size`. Sin embargo, en `blind_elf_main`, la dirección inicial y final vienen ambas dadas por el fichero ELF, por lo que se han definido las variables `ini` y `r` que corresponden a la dirección de memoria de la primera instrucción y de la última, respectivamente.

```
 /* Initializes some registers and
     allocates memory for mem variable */
if(!initialize())
    return 1;

/* ini is the first executable instruction's
 address, r is the last */
uint32_t *ini, r;

uint32_t res;

/* Reads the elf, loads it into the memory
     and pushes argc, argv and envp into the stack */
if(read_elf_file(argc, argv, envp, &ini,
```

```
                    &r, NULL, NULL, NULL)){
      perror("elf");
      goto exit;
   }
```

Por ello, cuando desensamblamos las instrucciones en `blind_elf_main`, en vez de indicar `size-eip` en el límite de bytes a desensamblar, indicamos `r - eip`.

```
   while (true){
      if(!cs_disasm(handle, &mem[eip], r-eip, eip, 1, &ins))
         goto exit;

      /* Check interrupts ???*/
      res = dispatcher(ins[0].mnemonic, &ins[0]);
```

Interface_elf_main Corresponde a la función que implementa la simulación de un programa ELF y con interfaz, ejecutando el programa paso a paso y permitiendo al usuario realizar ciertas operaciones para controlar el flujo del programa.

En primer lugar, al igual que las versiones sin interfaz, inicializamos el simulador y declaramos algunas variables. La variables declaradas a continuación ya han sido explicadas en el Apartado D.2. La variable `sheader` será un array que contendrá 2 valores por cada sección ejecutable, su dirección de memoria de inicio y su tamaño en bytes, mientras que la variable `cc` es el número de secciones ejecutables, por lo que `sheader` tiene tamaño `2*cc`.

Por otro lado, la variable `symbols` es un array que va a contener las direcciones de memoria en la que inician las funciones del fichero ELF, y la variable `strtab` será un array de cadenas que contendrá el nombre de dichas funciones, cuyos índices coinciden con `symbols`. La variable `ccc` será el número de funciones almacenadas en dichos arrays. Tanto `symbols` como `strtab` son inicializados con un puntero (da igual cual), ya que si estos valores son pasados a `read_elf_file` como nulos, asume que no hay nombres de funciones, pero si son valores distintos de `NULL`, asume que sí que hay nombres de funciones y los inicializa dentro. Es decir, son cargados con un valor cualquiera para que sean inicializados dentro de la función del *loader*, ya que si no, son ignorados.

```
int interface_elf_main(int argc, char *argv[], char *envp[]){
```

```
if(!initialize())
   return 1;

/* sheader will be an array of address and size
     of every executable section, and cc is the
     number of executable sections */
uint32_t *sheader, cc = 0;

/* Create pointers for functions names
      and addresses, and counter*/
uint32_t * symbols = &cc, ccc;
char * strtab = (char *)&cc;
```

A continuación, pasamos todas estas variables a la función del *loader*, que no sólo carga el ejecutable en memoria sino que prepara la pila con los valores necesarios y almacena los nombres y direcciones de memoria de las funciones, que serán usados más adelante. Luego imprimiremos una secuencia de caracteres de escape que borra la pantalla. También definiremos el *handler*, la instrucción para ejecutar paso a paso y el contador de instrucciones desensambladas.

```
/* Reads the elf, loads it into the memory and
     pushes argc, argv and envp into the stack */
if(read_elf_file(argc, argv, envp, &sheader,
         &cc, &symbols, &strtab, &ccc)){
   perror("elf");
   goto exit;
}

/* Clear screen and move pointer to (0,0) */
printf("\033[2J\033[H\033[3J");

/* Handler for disassembling */
csh handle;
/* ins is the step by step instruction to execute */
cs_insn *ins;
/* Number of instructions disassembled */
size_t count = 0;
```

Posteriormente, configuramos el *handler* y definimos el array de arrays de instrucciones. Será un array que en cada posición contenga un puntero al array de cada sección ejecutable. También crearemos otro array para guardar un registro de cuántas instrucciones tiene cada sección ejecutable.

```
 /* Sets arch */
if (cs_open(CS_ARCH_X86, CS_MODE_32, &handle) != CS_ERR_OK){
   perror("setarch");
   return 1;
}

/* Activate detail feature. Useful for
     obtaining operands and other info */
cs_option(handle, CS_OPT_DETAIL, CS_OPT_ON);

/* Array of pointer to sections instructions */
cs_insn **insns = calloc(cc, sizeof(cs_insn *));
/* Array containing accumulated number of instructions.
     Actual section + all instructions before */
int * counts = calloc(sizeof(int), cc);
```

Una vez tenemos dichas variables declaradas, procederemos a hacer el desensamblaje de las instrucciones de cada sección ejecutable. Primero obtenemos la dirección de memoria de inicio de la sección y su tamaño, a partir del array `sheader`, y desensamblamos en consecuencia. También almacenaremos el total acumulado de instrucciones desensambladas en el array `counts`.

```
 for (int i=0; i < cc; i++){
   uint32_t addr = sheader[2*i];
   uint32_t siz = sheader[2*i+1];
   uint32_t val = 0;
   if (addr && siz)
       val = cs_disasm(handle, &mem[addr], siz,
                       addr, 0, &insns[i]);
   count += ((val > 0)? val : 0);
   /* Obtain last instr number
         that belongs to this sh */
   counts[i] = count;
}
```

```
if (!count){
   printf("Failed to disassemble code\n");
   return 1;
}
```

A continuación, inicializamos la interfaz gráfica en la terminal.

```
 /* Initializes ncurses interaface */
init_interface();

/* getchar() does not need ENTER anymore */
init_raw_mode();
```

De manera adicional, deberemos reservar memoria para un array temporal que almacenará las strings (con las secuencias de escape de los colores) que corresponden a las instrucciones, tanto dirección de memoria como instrucción y operandos. Igualmente, realizaremos la misma operación con el array que almacena los nombres de las funciones que corresponden a cada instrucción a imprimir. Primero se reserva memoria para el array de punteros y luego para cada puntero a cadena.

```
 /* Allocates memory for pointer array. Its used to store
     the code lines to print. i.e "<0x08049752>:pop esi" */
char **lineas = malloc(2*rows * sizeof(char *));
if (lineas == NULL){
   perror("malloc");
   return 1;
}
char ** funcs = malloc((rows+1) * sizeof(char *));
/* Allocates memory for each pointer
     so it can store a string. */
for (int i = 0; i<rows; i++){
   /* Only for address */
   lineas[i*2]=malloc(ADDR_TXT_S);
   lineas[i*2+1]=malloc(MAX_STR);
   funcs[i] = malloc(FUNC_TXT_S);
}
```

También definiremos las variables que toma como argumentos la función `draw_screen()` del fichero `interface.c`, así como la variable `ch` que almacenará el caracter indicado por el usuario y el enfoque de la interfaz. Posteriormente buscaremos la instrucción inicial (aquella que coincide con el EIP tras cargar el programa en memoria) entre todas las secciones para establecerla como inicial y que debe ser resaltada.

```
 /* Char to get users char and scr_c to scroll
     instructions screen, scr_s to scroll stack screen,
     focus to focus on code or stack */
int ch = 0, old_ch = 0, scr_c = 0, scr_s = 0, focus = 0;
uint8_t found = 0;

/* Initialize scr_c to EIP instr*/
for(int i=0; i<cc;i++){
   uint32_t end = sheader[2*i+1];
   uint32_t ini = (!i)?0:counts[i-1];

   for (int j=0; j<end;j++){
      if (eip == insns[i][j].address){
         found = 1;
         scr_c = j+ini;
         scr_c = scr_c?scr_c-1:scr_c;
         break;
      }
   }
   if(found){
      break;
   }
}
```

Por último, antes de entrar en el bucle de ejecución, inicializaremos la tabla de *breakpoints*, y la variable que guarda el resultado temporal de cada instrucción.

```
/* Initialize breakpoints array */
uint32_t brkpts[MAX_BRK];

/* Initialize breakpoints counter */
uint8_t brk_ctr = 0;
```

```
/* Result of a instruction's execution */
uint32_t res = 0;
```

A continuación, entraremos en el bucle principal, cuya condición de salida será que el usuario haya presionado la tecla Q, es decir, `'q' == ch`. La variable `eip_ind` contiene un -º por defecto, pero se cambia a un índice de instrucción si se encuentra aquella cuya dirección de memoria coincide con el EIP, es decir, aquella que se ejecutará en el siguiente paso. La variable `exited` almacena en qué iteración se ha salido del bucle, ya que si es menor que el número total de líneas (`exited != rows-1`), se deben imprimir líneas vacías hasta llegar a `rows` para no desestructurar la interfaz.

Por otro lado, las variables `func`, `fbase`, `limit` y `strcount` son variables temporales usadas para buscar las funciones a las que pertenece cada instrucción. Esto se aprovecha de que las instrucciones que se impriman juntas serán, obligatoriamente, contiguas en el array de funciones, ya que las direcciones de memoria de estas están ordenadas.

```
/* While q is not pressed, what would quit the program */
while ('q' != ch){
      /* EIP index, if its negative, EIP is not
        visible on the current screen, due to scroll */
      int eip_ind = -1;
      int exited = -1;

      char * func;
      uint32_t fbase, limit, strcount;
```

Bucle para la obtención de texto Una vez definidas estas variables, iniciamos un bucle que para cada instrucción a mostrar, almacena su contenido en los arrays `lineas` y `funcs`. Este bucle es bastante extenso pero bien documentado y con variables autoexplicativas. La condición inicial, es decir, el número de veces que se repite el bucle será el mínimo entre el total de líneas reservadas para código en la interfaz (`rows`), y el total de instrucciones restantes hasta llegar al final (`count - scr_c`).

```
for (size_t i = 0; i < MIN(rows, count -scr_c+1); i++) {
```

De esta manera nos aseguramos que no se impriman más líneas de las que deberían ni más instrucciones de las que de verdad quedan. A

continuación, para cada iteración del bucle, buscaremos la sección ejecutable en la que reside la instrucción. Recordemos que `counts` es un array que contiene los índices máximos acumulados de cada sección ejecutable. Si por ejemplo tuvieramos 3 secciones, con 1, 4 y 7 instrucciones, el array `counts` equivaldría a `{1, 5, 12}`. Por lo que si el índice de una instrucción reside en la primera sección ejecutable cuyo valor es mayor que su índice de instrucción. Volviendo a nuestro ejemplo, si quisiéramos saber a qué sección corresponde la instrucción 4, miraríamos cual es el primer valor que es mayor que nuestro índice, que sería el valor 5 en la segunda posición del array, y por tanto residiría en la segunda sección ejecutable.

```
uint32_t index = i+scr_c;
uint32_t sh = 0;

/* Iterate through section headers */
for (int j=0; j<cc;j++){
    /* If sh last instruction number
            is greater than index */
    if(index < counts[j]){
        sh = j;
        break;
    }
}
```

Recordemos también que `scr_c` es el índice de instrucción correspondiente a la primera instrucción a imprimir, ya que está permitido deslizar por las instrucciones del programa, y que en la primera ejecución del bucle `while`, corresponde a la instrucción anterior al EIP. A continuación, para cada iteración del bucle exterior, obtendremos el número de instrucciones acumuladas de la sección anterior, ya que si se lo restamos a nuestro índice de instrucción actual, obtendremos el índice para la sección a la que pertenece.

Por ejemplo, si en el ejemplo anterior buscáramos la instrucción con índice global 7, necesitamos saber que lugar ocupa en su sección para poder acceder a ella, por lo que restaremos el número acumulado de instrucciones de la sección anterior (en nuestro caso de ejemplo era 5) para obtener el índice local de la instrucción en su sección. Por lo que obtendríamos que tendríamos que acceder al índice 2, por lo que haríamos `insns[sh][2]`. Esto lo hacemos para obtener la dirección de memoria de la instrucción y ver si coincide con el EIP, para establecer la variable `eip_ind` al valor actual de `i`.

```
/* Get previous section header last instruction index */
uint32_t prev = (!sh)?0:counts[sh-1];

uint32_t addr = insns[sh][index - prev].address;
if (addr == eip){
    eip_ind = i;
}
```

De manera adicional, una vez hemos obtenido la dirección de memoria de la instrucción, podemos saber a que función pertenece, y cuantos bytes está dentro de esa función. Por ejemplo, si tenemos una función `func1` que empieza en `0x08048000` y la siguiente comienza en `0x08048200`, si obtenemos una dirección de memoria `0x080480F8`, sabemos que está 0xF8 bytes dentro de la función `func1`, por lo que el texto de función a imprimir será `func1+0xF8`.

Para ello, primero encontraremos la función de la primera instrucción a imprimir (`i == 0`), ya que las funciones del resto de instrucciones serán la misma, o funciones contiguas. Recordemos que las posiciones `2*x +1` en el array `symbols` contienen las direcciones de memoria de las funciones. Si encontramos la función, guardaremos su dirección base, su límite (dirección de la siguiente función), el índice de la función (variable `strcount`) y la cadena de la función (variable `func`).

```
/* Find function name and address.  Only need to search
    the first one. Next instructions are going to be same
    function or a consecutive one.*/
if(i == 0){
    for (int k=0; k< ccc; k++){
        /* If current addr belongs to k function */
        if(symbols[2*k+1] <= addr &&
                addr < symbols[2*(k+1)+1]){

            /* Get pointer to Func String*/
            func = &strtab[symbols[2*k]];
            fbase = symbols[2*k+1];
            limit = symbols[2*(k+1)+1];
            strcount = k;
            break;
        }
    }
```

```
}
```

Si por el contrario, la instrucción que estamos estudiando no es la primera (`i != 0`), ejecutaremos lo siguiente. Es una comprobación de si la instrucción sigue en la misma función, se ha movido a la siguiente, o si ha avanzado a otra.

```
else{
    if (addr < limit){ /* Same function */
        /* Nothing changes*/
    }else if(strcount -1 >= 0 &&
            addr < symbols[2*(strcount)+1] &&
            addr >= symbols[2*(strcount-1)+1]){
        strcount--;
        limit = fbase;
        fbase = symbols[2*(strcount)+1];
        func = &strtab[symbols[2*strcount]];
    }else if(symbols[2*(strcount+1)+1] <= addr &&
        addr < symbols[2*(strcount+2)+1]){
        /* Next function */
        strcount++;
        fbase = limit;
        /* Get pointer to Func String*/
        func = &strtab[symbols[2*strcount]];
        limit = symbols[2*(strcount+1)+1];
    }else{  /* Other function (jmps, calls or rets)*/
        for (int k=0; k< ccc-1; k++){
            if(symbols[2*k+1] <= addr &&
                    addr < symbols[2*(k+1)+1]){
                /* Get pointer to Func String*/
                func = &strtab[symbols[2*k]];
                fbase = symbols[2*k+1];
                limit = symbols[2*(k+1)+1];
                strcount = k;
                break;
            }
        }
    }
}
```

Para finalizar el bucle principal, que tomaba la variable `i` desde 0 hasta el número de instrucciones a imprimir, fabricaremos las cadenas a imprimir, que pasaremos a `draw_screen()`, mediante los datos que hemos recolectado y la función `snprintf`, que imprime cadenas con formato en direcciones de memoria, en vez de la salida estándar.

Para las funciones, guardaremos el resultado en la posición `i` del array `funcs`, de manera que obtengamos `funcion+byte`, por ejemplo, `_start+12`. En líneas imprimiremos la dirección de memoria en base hexadecimal en las posiciones pares, y el contenido de la instrucción en las posiciones impares. También actualizaremos la variable `exited`, que sirve para saber cual ha sido el último valor del contador del bucle `i`.

```
        /* Add yellow to func name ?*/
        sprintf(funcs[i], "%s%s%s+%u",
               "\033[33m",func,"\033[0m", addr-fbase);

        snprintf(lineas[i*2], ADDR_TXT_S-1, "0x%08x", addr);
        snprintf(lineas[i*2+1], MAX_STR, "%.10s %.50s",
               insns[sh][index - prev].mnemonic,
               insns[sh][index - prev].op_str);
        exited = i;
    }
```

Posteriormente, comprobaremos la variable `exited` para ver si es necesario añadir líneas vacías. De esta manera borraremos el contenido de anteriores ejecuciones y dejaremos el array de cadenas en un estado consistente para su impresión.

```
    if (exited != rows-1){ /* No more instructions to show */
        while(exited != rows){
           snprintf(funcs[exited], 3," ");
           snprintf(lineas[exited*2], 3, " ");
           snprintf(lineas[exited*2+1], 3, " ");
           exited++;
        }
      }
```

Impresión y entrada de usuario Una vez tenemos preparados dichos arrays, ejecutaremos lo siguiente en cada iteración del bucle.

```
/* Draw registers, code and stack */
draw_screen(scr_s, scr_c, lineas, rows, eip_ind, funcs);

no_print:

/* Move pointer to last line and Gets user's option */
move(rows);

/* Disable echo from terminal so \n does not
      move the interface */
disable_echo();
fflush(stdout);

/* Get user's choice */
ch = getch();

/* Enable echo from terminal so user can see
      his input if his option needs getting input */
enable_echo();
fflush(stdout);

/* Clean stdin */
cleanv(rows-2, rows-2);

/* Set stdin cursor at first line of stdin */
cleanv(rows, rows);
```

Vemos que estamos imprimiendo el contenido actual en la terminal. Inmediatamente después hay una etiqueta `no_print` a la que saltan algunas opciones que no llegana ejecutar ninguna instrucción, por lo que no es necesario imprimir de nuevo la pantalla, ya que nada ha cambiado. A continuación ponemos la terminal en modo de lectura y leemos la tecla presionada por el usuario, para posteriormente dejarlo tal cual estaba justo antes.

Una vez tenemos la elección del usuario, procedemos a procesarla. Si la tecla es `'\n'`, significa que el usuario ha presionado la tecla `ENTER`, que traduciremos como repetir la última operación. También consideramos la posibilidad de que el caracter leído sea el `EOF`, si por ejemplo estamos utilizando una tubería para mandar payloads específicos, por lo que saltaríamos a `no_print` y volveríamos a leer un caracter.

```
/* Reading from pipe is giving EOF */
 if (EOF == ch){
    /* Do not print until != EOF so
           screen doesn't blink */
    goto no_print;
 }
 /* If user's choice is ENTER key */
 if('\n' == ch){
    /* Repeat last choice if ENTER is pressed */
    ch = old_ch;
 }
```

Una vez hemos comprobado estos dos casos aislados, entramos en una estructura if-else if para comprobar todos y cada uno de los caracteres posibles, mutuamente excluyentes.

Step La condición de esta operación es `'s' == ch`. Step se encarga de ejecutar una única instrucción, la apuntada por el registro EIP. Por ello, lo primero que deberemos hacer es desensamblar de nuevo la instrucción apuntada por el EIP, ya que si el usuario ha hecho *scroll* sobre las instrucciones, dicha instrucción puede no estar visible y por lo tanto no la tendríamos localizada. Si al tratar de resensamblar la instrucción, la función devuelve 0 (significando que no ha desensamblado nada), saltaremos a la etiqueta de error `exit`.

```
// If number of disasm instructions is 0.
if(!cs_disasm(handle, &mem[eip], 16, eip, 1, &ins))
    goto exit1;

 /* If returned from syscall, set
        the terminal on raw mode */
if (ins[0].bytes[0] == 0xCD){ /* INT imm8*/
        disable_raw_mode();
 }
```

De manera adicional, si la instrucción a ejecutar es una interrupción (`INT = 0xCD`), deshabilitaremos el modo *raw* de la terminal. Esto se debe a que si la interrupción desemboca en una llamada al sistema que utilice la salida o entrada estándar, necesitaremos la terminal en el modo habitual.

A continuación, limpiaremos la última línea de la terminal con `cleann()`, y moveremos el cursor de nuevo a la última línea con la función `movev()`. Para ejecutar la instrucción que hemos desensamblado anteriormente, llamaremos a la función `dispatcher()`, pasándole como argumentos en primer lugar el nemotécnico (`char *`), y en segundo lugar el puntero a la instrucción (`cs_insn *`). Por último, si la instrucción ha sido una interrupción, habilitaremos de nuevo el modo *raw* de la terminal, ya que la instrucción ya ha sido ejecutada. De esta manera, tendremos el resultados de la ejecución de la instrucción en la variable `res`.

```
cleann(rows, rows);
movev(rows);

 /* Check interrupts ???*/
 res = dispatcher(ins[0].mnemonic, &ins[0]);

 /* If returned from syscall, set the
            terminal on raw mode */
 if (ins[0].bytes[0] == 0xCD){ /* INT imm8*/
    enable_raw_mode();
 }
```

Posteriormente, liberaremos el espacio en memoria de la instrucción, ya que es la función `cs_disasm` la que reserva espacio para ella, y la próxima vez que se ejecute reservará espacio de nuevo para la siguiente instrucción. Por ello, si no liberamos la memoria de la instrucción temporal que utilizamos para ejecutar instrucciones, el espacio reservado crecería constantemente pero sin que lo usáramos ni que tuviéramos un puntero a esa región de la memoria.

```
\begin{verbatim}
/* Free ins */
cs_free(ins, 1);
ins = NULL;

/* Check if ins is syscall(1) (exit),
        to free memory before exiting */
if(0xdeadbeef == res)
    goto exit1;
```

Luego comprobaremos si el valor que ha retornado la instrucción (mediante la función `dispatcher()`) equivale al valor especial `0xdeadbeef`, que significa que ha ocurrido un error fatal o que s eha ejecutado la llamada al sistema `do_exit()`, por lo que el programa debe finalizar. Si esta comprobación es afirmativa, saltaremos a la etiqueta de error `exit1`.

Antes de finalizar la operación *Step*, dejaremos las variables globales necesarias con los valores adecuados. Por ello, debemos encontrar el índice de instrucción del nuevo EIP para la siguiente iteración, y que se imprima correctamente. Para ello, debemos darle nuevos valores a `scr_c` y `src_s`. Entre todas las secciones ejecutables buscaremos la instrucción cuya dirección coincida con el EIP y guardaremos su índice. Si el índice es 0, lo asignamos a `scr_c`, pero si es mayor que 0, estableceremos `scr_c` a la instrucción anterior al EIP, para que además de la instrucción a ejecutar también se vea la anterior.

```
if (!(eip < STACK_BOTTOM  && eip > STACK_TOP)){
    /* Find EIP and set it at the top of the screen */
    found = 0;
    for(int i=0; i<cc;i++){
        uint32_t end = sheader[2*i+1];
        uint32_t ini = (!i)?0:counts[i-1];

        for (int j=0; j<end;j++){
            if (eip == insns[i][j].address){
                found = 1;
                scr_c = j+ini;
                scr_c = scr_c?scr_c-1:scr_c;
                break;
            }
        }
        if(found){
            break;
       }
    }
}

/* Set top of stack at the top of stack window */
scr_s=0;
```

Por el contrario, `scr_s` la pondremos siempre a 0 para que cada vez que se ejecute una instrucción vuelva a la cima de la pila, por si se hubiera realizado alguna operación que la modifique.

Continue En el caso de esta operación, la condición es `'c' == ch`. *Continue* ejecuta instrucciones hasta que se encuentra con una instrucción cuya dirección de memoria está contenida en la tabla de *breakpoints*. Por ello, realiza las mismas operaciones que *Step*, solo que aplicado a la repetición. Por ello, toda la lógica de *Step*, excepto la búsqueda del nuevo EIP para asignarlo a `scr_c` se realiza dentro de un bucle con la siguiente condición de salida.

```
while (!contains(brkpts, brk_ctr, eip)){
    if(!cs_disasm(handle, &mem[eip], 16, eip, 1, &ins))
           goto exit1;

    ...

    if(0xdeadbeef == res)
           goto exit1;
}

if (!(eip < STACK_BOTTOM  && eip > STACK_TOP)){
    ...
}
scr_s = 0;
```

La función `contains()` ha sido explicada anteriormente. Busca en el buffer que se le pase, del tamaño que se le indique, el tercer parámetro, que es el valor a buscar. Por ello lo que estaríamos haciendo sería buscar en la tabla de *breakpoints* (`brkpts`), cuyo tamaño sería el número de breakpoints que hemos puesto hasta ahora (`brk_ctr` es 0 por defecto), un valor que coincida con el valor de EIP. Si encontramos dicho valor, es porque en la instrucción actual hay un *breakpoint*, por lo que debemos parar la ejecución y devolver el control al usuario.

No-Enter En el caso de esta operación, la condición es `'n' == ch`. La idea de *No-Enter*, es hacer más sencilla la ejecución del programa. Se encarga de ejecutar instrucciones hasta llegar a la instrucción inmediatamente siguiente a la inicial. Su objetivo es saltar la ejecución de funciones, ya que

las llamadas a funciones vuelven a la instrucción inmediatamente siguiente al ejecutar RET, y muchos bucles se basan en un JMP condicional. Por ello, con esta operación podemos evitar adentrarnos demasiado en un programa, o evitar bucles grandes.

Para ello, primero deberemos desensamblar la instrucción apuntada por EIP, para poder obtener su tamaño. Con su tamaño, si se lo sumamos al EIP, obtenemos la dirección de memoria de la instrucción siguiente, que será nuestra dirección de parada.

```
if(!cs_disasm(handle, &mem[eip], 16, eip, 1, &ins))
           goto exit1;
/* Get next instruction. Designed to avoid stepping
            into CALL instruction. */
uint32_t stop = eip + ins[0].size;
/* While doesnt hit a breakpoint, continue executing */
while (eip != stop){
    if(!cs_disasm(handle, &mem[eip], 16, eip, 1, &ins))
           goto exit1;

    ...

    if(0xdeadbeef == res)
           goto exit1;
}
if (!(eip < STACK_BOTTOM  && eip > STACK_TOP)){
    ...
}
scr_s = 0;
```

El interior del bucle *while* es el mismo que la operación *Continue*, es decir, todo lo ejecutado en *Step*.

Breakpoint La condición de esta operación es `'b' == ch`. El objetivo es añadir una dirección de memoria a la tabla de *breakpoints*. En primer lugar reservamos memoria para una *string* `str`, que es donde almacenaremos la respuesta del usuario. También inicializaremos dos variables a 0, `res` para el resultado del *parseo* de la dirección de memoria a partir de la cadena, y `c` para mostrar un mensaje de error si el formato es inválido.

La variable `c` cobra sentido al entender que la obtención de datos por parte del usuario se realiza dentro de un bucle *while*, y hasta que el formato

especificado por el usuario no sea correcto, se repetirá el bucle. Dicha repetición mostrará un mensaje de error cuando `c == 1`, es decir, a partir de la segunda iteración. La variable `dir` obtendrá la dirección de memoria indicada por el usuario.

Una vez el formato indicado ha sido correcto, y la dirección de memoria almacenada en la variable `dir`, la guardaremos en la posición del array `brkpts` que indique el contador `brk_ctr`.

```
char str[MAX_STR];
int res = 0, c = 0;
uint32_t dir;

/* While string received not matches
            the format 0x00000000 */
while(!res){
    /* Print asking string and get user's response */
    get_str("Breakpoint on direction : (0x00000000 format)",
            str, MAX_STR-1, c);
    /* Format check */
    res = sscanf(str, "0x%08x", &dir);
    /* Flag for adding "Wrong format" */
    c = 1;
}

/* Store breakpoint */
brkpts[brk_ctr]=dir;
```

Una vez hemos almacenado el nuevo *breakpoint*, limpiaremos las últimas 3 líneas de la terminal, ya que las hemos utilizado para imprimir cadenas y solicitar entrada al usuario. Informaremos de que el ha sido creado con éxito e incrementaremos el contador de *breakpoints*. Como el usuario podría escribir más de los permitidos, realizaremos la operación módulo sobre el contador, de manera que si se pasa del máximo, sobrescriba los primeros.

```
/* Clean stdin zone */
cleanv(rows-2, rows);
movev(rows-2);

/* Prints breakpont indexes from 1 to MAX_BRK*/
printf("Created breakpoint %u at 0x%08x", brk_ctr+1, dir);
```

```
/* Increment breakpoint counter */
brk_ctr++;
/* If counter overflows MAX, override the existing ones*/
brk_ctr %= MAX_BRK;
```

Delete La condición para esta operación es `'d' == ch`. El objetivo de *Delete* es borrar (o más bien inutilizar) un *breakpoint* existente. Para ello, realizaremos la misma lógica que con *Breakpoint*, pero solicitaremos un número con formato entero sin signo. Luego accederemos al array de *breakpoints* en la posición indicada por el usuario y cambiaremos su contenido por 0.

```
/* While string received not matches
            the format 0x00000000 */
while(!res){
    /* Print asking string and get user's response */
    get_str("Breakpoint number to remove : (0 format)",
                str, MAX_STR-1, c);
    /* Format check */
    res = sscanf(str, "%d", &dir);
    /* Flag for showing "Wrong format" */
    c = 1;
}
/* Transform breakpoint number to breakpoint index */
dir--;
/* Check overflow to avoid segmentation fault */
dir %= MAX_BRK;
/* Delete breakpoint */
brkpts[dir]=0x00000000;

/* Clear stdin and move pointer */
cleanv(rows-2, rows);
movev(rows-2);
```

Antes de acceder al array, decrementamos el número de *breakpoint*, ya que realmente el primer breakpoint (número 1) corresponde a la posición 0 del array. Posteriormente hacemos la operación de módulo antes de acceder, ya que si no estaríamos accediendo a una dirección de memoria que el usuario podría controlar. De esta manera conservamos el control de las direcciones a

las que puede acceder el usuario. Por último, borramos las últimas 3 líneas de la terminal, ya que las hemos usado para imprimir valores y recibirlos.

Show Hex La condición de esta operación es `'x' == ch`. La operación *Show Hex* se encarga de mostrar el contenido en hexadecimal de una dirección de memoria que indique el usuario. Para ello, obtendremos la entrada exactamente igual que en *Breakpoint*. Es decir, con formato de dirección de memoria hexadecimal. Una vez tenemos la dirección de memoria, limpiaremos las últimas líneas e imprimimos el resultado.

```
char str[MAX_STR];
int res = 0, c = 0;
uint32_t dir;

/* While string received not matches
                the format 0x00000000 */
while(!res){
    /* Print asking string and
                    get user's response */
    get_str("Address to dump content : (0x00000000 format)",
            str, MAX_STR-1, c);
    /* Format check */
    res = sscanf(str, "0x%08x", &dir);
    /* Flag for showing "Wrong format" */
    c = 1;
}

char txt[25];
snprintf(txt, 24, "0x%08x : 0x%08x",
            dir, *((uint32_t *)(mem +dir)));
cleanv(rows-2, rows);
movev(rows-2);
printf(" %s",txt);
```

Debemos convertir el puntero de la dirección de memoria a la que accedemos a un puntero de 32 bits, para que al desreferenciarlo nos devuelva una palabra de 4 bytes.

Show String La condición de esta operación es `'t' == ch`. Obtendremos una dirección de memoria exactamente igual que en *Show Hex*, pero

con la diferencia que en vez de imprimir el contenido de esa dirección de memoria, imprimiremos la cadena contenida en esa dirección de memoria.

```
char str[MAX_STR];
int res = 0, c = 0;
uint32_t dir;

/* While string received not matches
            the format 0x00000000 */
while(!res){
    /* Print asking string and get
                        user's response */
    get_str("Address to dump string : (0x00000000 format)",
                str, MAX_STR-1, c);
    /* Format check */
    res = sscanf(str, "0x%08x", &dir);
    /* Flag for showing "Wrong format" */
    c = 1;
}
cleanv(rows-2, rows);
movev(rows-2);
printf(" 0x%08x : %s",dir, mem+dir);
```

Para ello, debemos imprimir el contenido de la dirección `mem+dir` mediante el modificador `%s` de `printf()`.

Focus Code La condición de esta operación es `KEY_LEFT == ch` o `'4' == ch`. El objetivo es establecer la variable `focus` a 0, es decir, al código. Que el enfoque esté en el código, implica que ciertas operaciones, como `find` o *scrollear* afecten solamente al código.

```
cleann(rows, rows);
movev(rows);
/* Switch scroll focus to Code */
focus = 0;
goto no_print;
```

Focus Stack La condición de esta operación es `KEY_RIGHT == ch` o `'6' == ch`. El objetivo es establecer la variable `focus` a 1, es decir, a la pila.

Que el enfoque esté en la pila, implica que ciertas operaciones, como `find` o *scrollear* afecten solamente a la pila.

```
cleann(rows, rows);
movev(rows);
/* Switch scroll focus to Code */
focus = 0;
goto no_print;
```

Scroll Up La condición de esta operación es `KEY_UP == ch` o `'8' == ch`. Modifica las variables `src_c` o `scr_s` en función de si el enfoque es 0 o 1 (código o pila), respectivamente. La variable `scr_c` debe ser mayor o igual que 0. La variable `scr_s` debe ser igual o mayor que 0 también. Si la variable a modificar fuera 0, no podemos decrementarla así que saltaremos a la etiqueta `no_print`, que equivale a no realizar ningún cambio.

```
 cleann(rows, rows);
movev(rows);
/* Focus is set on Code and scr_c doesnt underflow */
if (focus == 0){
    if (scr_c > 0){
        /* Scroll up */
        scr_c--;
    }else{
        goto no_print;
    }
}
/* Focus is set on Stack and scr_s doesnt underflow */
else{
    if(scr_s > 0){
        /* Scroll up */
        scr_s--;
    }else{
        goto no_print;
    }
}
```

Scroll Down La condición de esta operación es `KEY_DOWN == ch` o `'2' == ch`. Modifica las variables `src_c` o `scr_s` en función de si el enfoque es 0 o 1 (código o pila), respectivamente. La variable `scr_c` debe ser menor

que la variable `count`. La variable `scr_s` debe ser menor que el número de valores en pila. Si la variable a modificar excediera el límite, no podemos incrementarla así que saltaremos a la etiqueta `no_print`, que equivale a no realizar ningún cambio.

```
cleann(rows, rows);
movev(rows);
/* Focus is set on Code and scr_c doesnt overflow */
if (focus == 0){
    if( scr_c+1 < count){
        /* Scroll down */
        scr_c++;
    }else{
        goto no_print;
    }
}
/* Focus is set on Stack, overflow checked */
else{
    if( scr_s+1 < (STACK_BOTTOM - (int)esp)/4 ){
        /* Scroll down */
        scr_s++;
    }else{
        goto no_print;
    }
}
```

Find La condición de esta operación es `'f' == ch`. Establece o bien una instrucción o una dirección de pila en la primera posición de sus respectivas ventanas. Si el enfoque está puesto en la ventana de código, si buscamos una dirección de memoria asociada a una instrucción, de existir se establecerá la primera de la ventana. Si por el contrario, buscamos una dirección de pila con el enfoque puesto en el código, nada ocurrirá. Por otro lado, si el enfoque está puesto en la pila, solo podremos buscar direcciones en pila.

Para ello, primero obtenemos la dirección de memoria por parte del usuario, de igual manera que con opciones anteriores.

```
char str[MAX_STR];
int res = 0, c = 0;
uint32_t dir;
```

```
char txt[MAX_STR];

focus?snprintf(txt, MAX_STR,
    "Stack address to lookup : (0x00000000 format)"):
snprintf(txt, MAX_STR,
    "Code address to lookup : (0x00000000 format)");

while(!res){
    /* Print asking string and get user's response */
    get_str(txt, str, MAX_STR-1, c);
    /* Format check */
    res = sscanf(str, "0x%08x", &dir);
    /* Flag for showing "Wrong format" */
    c = 1;
}
```

Una vez hemos salido del bucle *while*, y obtenido una dirección de memoria válida, comprobamos en qué ventana reside el enfoque, para realizar el cambio de la variable `scr_c` o `scr_s`, según corresponda. Si la variable es de pila, podemos averiguarla realizando una operación. En cambio, si es de código, deberemos iterar por las secciones ejecutables y localizar la dirección de memoria que coincida.

```
if (focus){
    if (dir < STACK_BOTTOM && dir > STACK_TOP){
        scr_s = (int)((dir-esp)/4);
    }
}else{
    found = 0;
    for(int i=0; i<cc;i++){
        uint32_t end = (!i)?counts[i]:counts[i]-counts[i-1];
        uint32_t ini = (!i)?0:counts[i-1];

        for (int j=0; j<end;j++){
            if (dir == insns[i][j].address){
                found = 1;
                scr_c = j+ini;
                break;
            }
        }
```

```
            if(found){ break; }
        }
    }
    cleanv(rows-2,rows);
```

Si encontramos la dirección, saldremos del bucle, y tanto si hemos encontrado la dirección o no, sea de pila o de código, borramos las últimas líneas ya que las hemos utilizado para solicitar la entrada del usuario.

Interface_raw_main Corresponde a la función que implementa la simulación de un programa *Raw* y con interfaz, ejecutando el programa paso a paso y permitiendo al usuario realizar ciertas operaciones para controlar el flujo del programa.

Es exactamente igual a `interface_elf_main()`, con la pequeña diferencia que no se obtienen los nombres de las funciones del programa, y que las instrucciones se almacenan en un único array continuo de instrucciones, es decir, un `cs_insn *`, a diferencia de la versión ELF donde se almacenaba en un `cs_insn **` que contenía varios `cs_insn *`.

Compilación, instalación y ejecución del proyecto

Este proceso está explicado con detalle en el Apartado E.1.

Pruebas del sistema

Para realizar las pruebas del simulador, se han creado una serie de scripts que permiten realizar las pruebas de manera automática, generando un informe donde se produce algún hecho atípico. Para ello, se ejecuta un mismo programa con el *debugger* de GNU, GDB, pero modificándole algunos parámetros. Algunos de estos parámetros son, poner a 0 las direcciones de pila del vector auxiliar que corresponden a la VDSO. La VDSO (Virtual Dynamic Shared Object) [5] es una librería que permite realizar algunas llamadas al sistema de manera diferente, evitando pasarle el control al Kernel, lo que las hace más rápidas. Estos parámetros han de ser modificados ya que el simulador no emplea dicha librería dinámica.

Poniendo a 0 las direcciones del vector auxiliar que corresponden a la VDSO, nos aseguramos que cuando el programa requiera hacer una llamada al sistema de esta índole, la haga pasándole el control al Kernel, y por lo tanto el programa ejecutado por GDB y por nuestro simulador vayan por el

mismo camino, ejecutando las mismas instrucciones, y esperando el mismo resultado.

```
gdb.execute("break _start")
gdb.execute("run")
gdb.execute("set *0xffffd310=0x0")
gdb.execute("set *0xffffd318=0x0")
```

De igual manera, es necesario alinear las pilas de dichos dos programas hasta el punto que coincidan byte a byte. Esto se debe a que si las variables de entorno difieren, también variarán sus direcciones de memoria, y las instrucciones que se ejecutan al procesarlas, por lo que el flujo del programa dejará de ser el mismo. Para asegurar que ambos programas usan las mismas variables de entorno, debemos eliminar aquellas añadidas por GDB o que no vayan a coincidir (`LINES`, `COLUMNS`, `_`, `OLDPWD`|).

```
unset environment _
unset environment LINES
unset environment COLUMNS
unset environment OLDPWD
```

Una vez conseguimos alinear ambas pilas, el flujo del programa es a grandes rasgos el mismo, pudiendo diferir en resultados de llamadas al sistema (aquellas que devuelven el PID o algo relacionado por ejemplo) o aquellas que dejen indeterminada alguna bandera del registro EFLAGS, pudiendo estar encendida en un sitio y apagada en otro.

Este proceso está explicando con detalle en uno de los videos disponibles en el Apéndice G.1.

D.3. Versión para sistemas Windows

Esta versión corresponde a la rama `win32` del Repositorio del proyecto, que es aquella orientada a la ejecución en sistemas Windows mediante una interfaz gráfica de usuario basada en .NET y Windows Forms. El contenido de este repositorio se plantea para ser compilado como una librería DLL, que sea utilizada por la interfaz gráfica en cuestión, que está disponible en el Repositorio de la Interfaz Gráfica.

Estructura de directorios

- **Directorio capstone**: contiene las cabeceras de la librería *Capstone*, necesarias a la hora de compilar el DLL.
- **Directorio lib**: contiene las cabeceras de los ficheros de código fuente del simulador (extensión .h).
- **Directorio src**: contiene los ficheros de código fuente que implementan el simulador (extensión .c).
- **README.md**: fichero que contiene una breve descripción y explicaciones acerca de su instalación y uso.
- **LICENSE**: fichero que contiene el texto de la licencia del simulador.
- **NOTICE**: fichero que contiene los textos de las licencias de los otros proyectos utilizados.
- **compile.bat**: script que permite compilar la librería DLL.

Manual del programador - librería DLL

En esta rama, se ha conservado la mayor parte del código de la rama principal. Sin embargo, debido a la limitación técnica por la que no se pueden ejecutar las llamadas al sistema en Windows, ya que sería necesaria otra traducción distinta, que está fuera del objetivo del proyecto, han sido podados algunos módulos.

Los módulos eliminados en esta rama son `proc`, `trad_syscall`, `syscall` y `typesi386`. Los módulos `interrupts` y `instr` ha sido modificado ligeramente para esta rama, y ha sido creado el módulo `main`.

interrupts.c

Este fichero ya no ejerce como interfaz entre las instrucciones y las interrupciones (y por ende las llamadas al sistema), sino que, como se sobrentiende que no se deben ejecutar interrupciones en la rama de Windows, cualquier llamada a la función `int_dispatcher()` desembocará en un valor de retorno `0xf4`, que indica finalización del programa.

```
uint32_t int_dispatcher(uint8_t v, uint32_t *eax,
        uint32_t *ebx, uint32_t *ecx, uint32_t *edx,
```

```
        uint32_t *esi, uint32_t *edi, uint32_t *ebp){
    /* No syscalls suport on win32. Exit immediatly. */
    return  0xF4;
}
```

instr.c

En el fichero `instr.c` se han realizado algunas modificaciones menores. Estas modificaciones tienen como objetivo cambiar lo que espera el módulo `proc` por lo que espera el módulo `main`. Por ello, las secciones de código o instrucciones que devolvían el valor especial `0xdeadbeef`, ahora pasan a devolver `0xf4`, que es el valor de finalización del programa. Este valor no se ha elegido al azar sino que corresponde con el *bytecode* de la instrucción `HLT`, que detiene el simulador.

```
int hlt_i (cs_insn *insn){
    return 0xF4;
}
```

Por otro lado, han sido eliminadas las secciones que implementaban las funcionalidades de las técnicas de mitigación NX, ASLR o debug.

main.c

En cuanto al fichero `main.c`, se han definido funciones auxiliares que son necesarias en el programa que implementa la interfaz gráfica en Windows.

Macro de Exportación Para que puedan estar disponibles cuando compilemos el código como un DLL, debemos indicar un macro específico. Debido a que el macro es muy extenso como para añadirlo en cada función, se ha creado un segundo macro llamado `EXPORT` que equivale al macro en cuestión. Si no estamos en Windows (el macro `_WIN32` no está definido), decimos que el macro `EXPORT` no significa nada, para que no de error al compilar.

```
#ifdef _WIN32
    #define EXPORT __declspec(dllexport)
#else
    #define EXPORT
#endif
```

Definición de variables Para que el simulador pueda funcionar, necesitamos definir una serie de variables auxiliares que serán estáticas al DLL compilado, se mantendrá su valor mientras se esté usando la librería. En primer lugar, definimos referencias a los registros y a la memoria, para poder usarlos en el fichero actual.

```
EXPORT extern uint32_t eax, edx, ecx, ebx, ebp,
    esp, eip, esi, edi, eflags, cs, ds, es, gs,
    fs, ss;
EXPORT extern uint8_t * mem;
```

A continuación, las variables necesarias para desensamblar instrucciones y almacenar los resultados y poder imprimirlos.

```
/* Handler for disassembling */
csh handle;
/* insn is the pointer to instructions array,
ins is the step by step instruction to execute */
cs_insn *insn, *ins;
```

De igual manera, `v` será el número de direcciones de pila a mostrar, `s` será el número de instrucciones desensambladas. La variable `caddrs` es un array que contiene las direcciones de memoria de las instrucciones, en función de su índice, que también coincide con `instrstrs`, que contiene la cadena de la instrucción (nemotécnico + operandos).

```
int v,s;
uint32_t * caddrs;
char ** instrstrs;
```

Por último, definimos la tabla de contadores y el índice de contadores usados.

```
uint8_t nusedbrks = 0;
uint32_t breakpoints[NBREAKS];
```

Getters Una vez podemos indicar que son funciones a exportar, definiremos aquellas que sean estrictamente necesarias. En primer lugar, la interfaz va a necesitar consultar los valores de los registros para poder mostrarlo en la interfaz.

```
EXPORT uint32_t get_eax(){ return eax; }
EXPORT uint32_t get_ebx(){ return ebx; }
EXPORT uint32_t get_ecx(){ return ecx; }
EXPORT uint32_t get_edx(){ return edx; }
EXPORT uint32_t get_edi(){ return edi; }
EXPORT uint32_t get_esi(){ return esi; }
EXPORT uint32_t get_ebp(){ return ebp; }
EXPORT uint32_t get_esp(){ return esp; }
EXPORT uint32_t get_eip(){ return eeip; }
EXPORT uint32_t get_eflags(){ return eflags; }
```

También definiremos *getters* para los selectores de segmento.

```
EXPORT uint32_t get_cs(){return cs;}
EXPORT uint32_t get_es(){return es;}
EXPORT uint32_t get_ds(){return ds;}
EXPORT uint32_t get_fs(){return fs;}
EXPORT uint32_t get_gs(){return gs;}
EXPORT uint32_t get_ss(){return ss;}
```

Por último, definimos *getters* para valores hexadecimales o cadenas asociados a una dirección de memoria. Corresponden a los casos de uso *Show-Hex* y *Show-String*.

```
EXPORT uint32_t show_hex_opt(uint32_t dir){
    return *((uint32_t *)(mem + dir));
}

EXPORT char * show_string_opt(uint32_t dir){
    return ((char *)(mem + dir));
}
```

Breakpoints Para poder permitir crear breakpoints desde la interfaz gráfica, debemos exponer una función en el DLL que lo haga. Asumiremos que ya se ha comprobado la validez de la dirección de memoria desde el programa que emplea el DLL. Crearemos una función que pueda crear el *breakpoint*, y otra que pueda borrarlos.

```
EXPORT int breakpoint_opt(uint32_t adr){
    breakpoints[nusedbrks] = adr;
    nusedbrks++;
    uint8_t store = nusedbrks;
    nusedbrks %= NBREAKS;

    return store;
}

EXPORT int delete_opt(uint8_t ind){
    breakpoints[ind % NBREAKS] = 0;
    return 0;
}
```

Inicialización La inicialización se realiza mediante la función `init_sim()`. Que en primer lugar inicializa los registros a 0. A continuación, inicializamos la memoria, y cargamos el programa en memoria.

```
v = vv;
initialize();
uint32_t nbytes = read_raw_file(f);
```

Posteriormente, desensamblamos las instrucciones.

```
/* Sets arch */
if (cs_open(CS_ARCH_X86, CS_MODE_32, &handle)
                    != CS_ERR_OK){
    perror("setarch");
    return 1;
}

/* Activate detail feature. Useful for
    obtaining operands and other info */
cs_option(handle, CS_OPT_DETAIL, CS_OPT_ON);
```

```
/* Store number of disassembled instructions */
s = disassemble(nbytes);
```

Por último, cargamos los valores de las instrucciones en los arrays anteriormente mencionados.

```
caddrs = calloc(s, sizeof(uint32_t));
instrstrs = calloc(s, sizeof(char *));
for(int j=0; j<s; j++){
    instrstrs[j] = calloc(NSTR, sizeof(char));
}
```

Una vez ejecutada, también se debe ejecutar la función `get_code_content()`, que rellena dichos arrays con su dirección de memoria y su contenido.

```
EXPORT void get_code_content(int s){
    /* Fill array with instructions values */
    for(int i=0; i<s; i++){
        caddrs[i] = insn[i].address;
        snprintf(instrstrs[i], NSTR-1,"%s %s",
                insn[i].mnemonic, insn[i].op_str);
    }
}
```

Ejecución A continuación, definiremos las funciones para los casos de uso *Step*, *Continue* y *No-Enter*, que permitirán ejecutar instrucciones según desee el usuario.

```
EXPORT int step_opt(){
    int p = cs_disasm(handle, &mem[eip],
                MAX_INSTR_SIZE, eip, 1, &ins);
    return dispatcher(ins->mnemonic, ins);
}

EXPORT int continue_opt(){
    uint32_t ret = 0;
    while (!contains(breakpoints, nusedbrks,
            eip) && ret != 0xF4){
```

```
            ret = (uint32_t)step_opt();
        }
        return ret;
    }

    EXPORT int no_enter_opt(){
        int p = cs_disasm(handle, &mem[eip],
                    MAX_INSTR_SIZE, eip, 1, &ins);
        uint32_t stop = eip + ins->size, ret = 0;
        while (eip != stop && ret != 0xF4){
            ret = step_opt();
        }
        return ret;
    }
```

Cierre Cuando se sale del programa, o cuando el usuario lo indique, cerraremos el simulador. Para ello, liberaremos la memoria que hemos reservado para la simulación del programa.

```
    EXPORT int exit_sim(){
        free(mem);

        cs_free(insn, s);
        free(caddrs);
        free(instrstrs);

        return 0;
    }
```

Compilación

En este apartado se va a proceder a detallar el proceso de compilación, instalación y ejecución del simulador en su variante Windows. Este proceso está explicado con detalle en el Apéndice G.2.

Este paso es opcional ya que se puede descargar el DLL desde las *releases* del Repositorio del proyectoPara compilar la librería DLL requerida para la ejecución del simulador, deberemos instalar o bien Microsoft Visual Studio, o simplemente las herramientas de Visual Studio. Con ello obtendremos el programa compilador `cl`, que es el que vamos a emplear para llevar a cabo la compilación.

Una vez tengamos el programa, debemos clonar la rama `win32` del proyecto, de manera que en una carpeta tengamos los directorios `src`, `lib` y `capstone`. Una vez preparado el código fuente, deberemos instalar en la misma carpeta la librería `capstone.lib`, disponible en el repositorio de Capstone para Windows.

Una vez disponemos de todo lo necesario, compilaremos el DLL con el siguiente comando.

```
cl /LD .\src\*.c /Fe:i386-engine.dll
    /link /LIBPATH:.\ capstone.lib
```

Esto nos dejará con un fichero llamado `i386-engine.dll`, que es exactamente el que buscamos.

D.4. Manual del Programador de la GUI con Windows Forms

Para no engordar demasiado este documento de Anexos, el código de la interfaz gráfica de usuario para Windows (aquella que emplea el DLL del simulador) ha sido mostrado mediante un video explicativo disponible en el Apéndice G.2. Este video muestra a grandes rasgos como se ha programado la interfaz, y como ésta utiliza el DLL del simulador para poder ofrecer al usuario la experiencia de simulación del programa que desee.

Apéndice E

Documentación de usuario

E.1. Introducción

El actual proyecto tiene como objetivo desarollar un simulador que permita ejecutar programas del i386. Dicho objetivo ha sido alcanzado mediante la programación de un núcleo del simulador, que ha permitido desarollar versiones del simulador para Linux y para Windows.

La versión de Linux implementa de manera nativa el núcleo del simulador, compilándose de manera conjunta a la vez, mientras que la versión de Windows requiere realizar algunas modificaciones y compilarlo como una librería dinámica para Windows, ofreciendo una interfaz alternativa mediante Windows Forms y .NET. A continuación se van a detallar los procesos de usuario relativos a ambos entornos, tanto instalación como ejecución de programas.

E.2. Manual de usuario en entorno Linux

Para esta versión del simulador, el entorno esperado es una máquina con un sistema operativo basado en el Kernel de Linux. Una arquitectura x86_64 permite de manera adicional la ejecución de llamadas al sistema, de manera que en otras arquitecturas la ejecución está limitada a instrucciones. Existen videos explicativos de los siguientes apartados en el Apéndice G.

Instalación

Para llevar a cabo la instalación en Linux, primero debemos asegurarnos de tener `git` instalado, o bien descargar el repositorio manualmente desde

GitHub. Desde la terminal, podemos clonar el repositorio con el siguiente comando.

```
git clone https://github.com/miguelgonpam/Sim-i386-32bit/
cd Sim-i386-32bit/
```

Una vez tenemos clonado el repositorio, debemos instalar la librería de C de Capstone, para poder recompilar el simulador, así como `gcc` si todavía no lo tenemos instalado.

```
sudo apt install libcapstone-dev gcc -y
```

Una vez tenemos instalado la librería Capstone y `gcc`, podemos compilar el programa. Para compilar podemos ejecutar el script `compile.sh` o bien hacerlo a mano.

```
# Cualquiera de las dos opciones
gcc ./src/*.c -o proc -lcapstone -lc

./compile.sh
```

Con ello, ya tendremos el simulador compilado y preparado para su uso.

Compilación de programas

El simulador toma como argumento imprescindible el nombre de un ejecutable (junto con sus opciones). Por ello, antes de ejecutar el simulador, deberemos compilar algunos programas para poder simularlos. Tenemos tres alternativas.

- Compilar programas escritos en C.
- Compilar programas escritos en ensamblador.
- Crear programas escribiendo *bytecode* a mano.

Compilación de programas escritos en C

Para poder compilar estos programas a *bytecode* que entienda el i386, deberemos crear nuestro propio *Cross Compiler*, de manera que sea un programa que se ejecute sobre la arquitectura de nuestra máquina pero que obtenga ejecutables de i386. Para ello vamos a ayudarnos de otro proyecto, la librería musl, que implementa la *libc* de manera más ligera.

En primer lugar, clonaremos el proyecto *musl*, y crearemos el fichero de configuración. Posteriormente lo compilaremos, lo que nos otorgará el ejecutable que estamos buscando, junto con otras herramientas útiles.

```
git clone https://github.com/richfelker/musl-cross-make
cd musl-cross-make
```

Una vez dentro del directorio, creamos un fichero llamado `config.mak` y escribimos el siguiente texto en él.

```
TARGET = i386-linux-musl
OUTPUT = /usr/local/gcc-i386
COMMON_CONFIG += --disable-nls
GCC_CONFIG += --disable-libquadmath --disable-decimal-float
GCC_CONFIG += --disable-libitm
GCC_CONFIG += --disable-fixed-point
GCC_CONFIG += --disable-lto
```

Una vez tengamos preparado el fichero de configuración ejecutaremos los siguientes comandos, que instalan las librerías y cabeceras necesarias, compila, e instala el resultado.

```
# Instalar cabeceras y librerías
sudo apt install build-essential libgmp-dev libmpfr-dev
    \ libmpc-dev gcc-multilib libc6-dev-i386

# Compilar
make -j$(nproc)

# Instalar
sudo make install
```

Una vez ejecutados todos los comandos, encontraremos los binarios en `/usr/local/gcc-i386/bin`, o en la subcarpeta `bin` de la ruta que hayamos indicado como `OUTPUT` en el fichero de configuración. El fichero que nos incumbe es `i386-linux-musl-gcc`. Podemos añadirlo al `PATH` o bien crear un enlace simbólico al ejecutable en nuestra carpeta de trabajo.

```
ln -s /usr/local/gcc-i386/bin/i386-linux-musl-gcc ~/gcc-i386
```

Una vez tengamos el *Cross Compiler*, podremos compilar el programa C que queramos, pero siempre con las opciones de compilación estática y sin PIE.

```
~/gcc-i386 programa.c -o programa.elf -static -no-pie
```

Esto resultará en un fichero `programa.elf` que podremos ejecutar con el modo ELF del simulador.

Compilación de programas escritos en ensamblador

Para poder compilar programas escritos en ensamblador, necesitaremos instalar una herramienta de terceros llamada `NASM` [6]. Esta herramienta es un proyecto comunitario, distribuido bajo licencia BSD de 2 cláusulas, y muy usado en proyectos de sistemas operativos o proyectos a bajo nivel en general. Permite compilar binarios a partir de ensamblador, para 16 bits, 32 bits o 64 bits.

Para utilizar esta herramienta, primero deberemos generar nuestro fichero con instrucciones ensamblador, al que podremos llamar `programa.asm`. Un fichero de este estilo debe parecerse a lo siguiente. El macro `BITS 32;` indica al compilador que queremos generar código de 32 bits para nuestro i386.

```
BITS 32;

inicio:
    mov eax, 0x11223344
    push eax
    pop ebx
    xor eax, eax
    inc eax
    int 0x80
```

Una vez tenemos el programa en ensamblador, deberemos utilizar `nasm` para compilarlo a fichero binario. El parámetro `-f` indica el formato de salida que queremos, y el parámetro `-o` indica el nombre del fichero de salida.

```
nasm -f bin programa.asm -o programa.bin
```

Esto resultará en un fichero `programa.bin` que podremos ejecutar con el modo RAW del simulador.

Creación de programas escribiendo *bytecode*

La creación de programas escribiendo *bytecode* es un proceso más complejo ya que conlleva un conocimiento más profundo de la naturaleza del i386. Para poder llevarla a cabo, deberemos consultar recurrentemente el Manual del programador [4], donde se indica a qué bytes específicos equivale cada instrucción y en qué contexto.

Por ejemplo, si quisiéramos escribir el *bytecode* de la instrucción `ADD`, para sumarle un valor al registro `EAX`, deberíamos consultar la página del manual asociada a la instrucción `ADD`, y una vez en ésta, seleccionar la opción que se ajusta a nuestro contexto.

```
Opcode     Instruction     Description

04 ib      ADD AL,imm8       Add immediate byte to AL
05 iw      ADD AX,imm16      Add immediate word to AX
05 id      ADD EAX,imm32     Add immediate dword to EAX
80 /0 ib   ADD r/m8,imm8     Add immediate byte to r/m byte
81 /0 iw   ADD r/m16,imm16   Add immediate word to r/m word
81 /0 id   ADD r/m32,imm32   Add immediate dword to r/m dword
83 /0 ib   ADD r/m16,imm8    Add sign-extended immediate byte
                                to r/m word
83 /0 ib   ADD r/m32,imm8    Add sign-extended immediate byte
                                to r/m dword
00 /r      ADD r/m8,r8       Add byte register to r/m byte
01 /r      ADD r/m16,r16     Add word register to r/m word
01 /r      ADD r/m32,r32     Add dword register to r/m dword
02 /r      ADD r8,r/m8       Add r/m byte to byte register
03 /r      ADD r16,r/m16     Add r/m word to word register
03 /r      ADD r32,r/m32     Add r/m dword to dword register
```

Por ejemplo, como queremos sumarle un valor al registro EAX, deberemos usar la siguiente alternativa, e indicar el valor inmediato en *Little Endian*, es decir, con el byte menos significativo primero, por lo que `0x00001001` equivaldría a `0x01 0x10 0x00 0x00`.

```
05 id     ADD EAX,imm32    Add immediate dword to EAX
```

Por lo que la instrucción completa sería:

```
0x05 0x01 0x10 0x00 0x00
```

Este *payload* podemos escribirlo en un fichero `.bin` para posteriormente ejecutarlo en el simulador. Podemos hacerlo o bien utilizando una herramienta como `hexedit`, que permite escribir byte a byte en un fichero, o escribiendo los bytes con `printf` y redireccionándolo a un fichero.

```
printf "\x05\x01\x10\x00\x00" > programa.bin
```

Con esto obtendremos un fichero `.bin` que podremos ejecutar en el simulador con su modo RAW.

Ejecución

En cuanto a la ejecución del programa, una vez compilado y con el programa a simular preparado, deberemos ejecutar el programa e indicar las opciones que necesitemos. Las opciones disponiles son las siguientes.

Flag	**Opción completa**	**Explicación**
-i	--interface	Ejecuta el programa con interfaz gráfica, paso a paso.
-n	--normal	Ejecuta el programa como si se ejecutara desde la terminal, de una sola vez.
-e	--elf-mode	Ejecuta el programa de manera que simule ficheros ELF.
-r	--raw-mode	Ejecuta el programa de manera que simule ficheros RAW.
-d	--debug	Activa el debugging, que almacena en un fichero log el estado de los registros a cada instrucción ejecutada.
-x	--nx	Activa la técnica de mitigación NX.
-a	--aslr	Activa la técnica de mitigación ASLR.
-t <adr>	--stack-top <adr>	Permite modificar la dirección de memoria donde acaba la pila (dirección baja de la pila).
-b <adr>	--stack-bottom <adr>	Permite modificar la dirección de memoria donde empieza la pila (dirección alta de la pila).
-h	--help	Muestra la ayuda del programa.

Tabla E.1: Argumentos del simulador en la versión Linux.

De todas estas opciones, solo dos son obligatorias. Se deben especificar de manera obligatoria una opción de formato ejecutable y una opción de interfaz, por lo que las alternativas posibles son `-ie`, `-ir`, `-ne` y `-nr`. Junto con estas opciones debe indicarse de manera obligatoria el nombre de un fichero acorde al formato ejecutable seleccionado. Si hemos seleccionado la opción `-e` debemos indicar un fichero ELF, y si no, con `-r` debemos indicar un fichero RAW.

De igual manera las opciones `-b` y `-t` son incompatibles con la opción `-a`. Algunas combinaciones de opciones podrían ser:

```
./proc -ie programa.elf
./proc -ir programa.bin
./proc -ne programa.elf
./proc -iet 0xfffdc000 -b 0xffffe000 programa.elf
./proc -iea programa.elf
```

```
./proc -nedt 0xfffdc000 -b 0xffffe000 programa.elf
./proc -ieax programa.elf
```

Si hemos seleccionado la opción `-n`, el programa no requiere de más interacción por nuestra parte. Sin embargo, con el modo `-i`, debemos manipular la interfaz para llevar a cabo la ejecución paso a paso del programa. De nuevo, volvemos a tener una lista de opciones.

Tecla	**Nombre**	**Explicación**
S	Step	Ejecuta únicamente la siguiente instrucción
B	Breakpoint	Crea un punto de parada en el programa, en una dirección de memoria específica
D	Delete breakpoint	Borra un punto de parada existente
C	Continue	Ejecuta tantas instrucciones como sea necesario hasta encontrar un punto de parada o finalizar el programa
N	No-Enter	Ejecuta tantas instrucciones como sea necesario hasta llegar a la instrucción inmediatamente siguiente (Evitar JMP's o CALL's)
↑	Scroll up	Se mueve hacia arriba en la ventana con el enfoque activo
↓	Scroll down	Se mueve hacia abajo en la ventana con el enfoque activo
←	Set focus to code	Pone el enfoque en la ventana de código
→	Set focus to stack	Pone el enfoque en la ventana de pila
F	Find address	Encuentra una dirección de memoria en la ventana con el enfoque activo
X	Show Hex	Muestra el valor hexadecimal de una dirección de memoria que se le indique
T	Show String	Muestra la cadena almacenada en una dirección de memoria que se le indique

Tabla E.2: Opciones de la interfaz del simulador en la versión Linux.

E.3. Manual de usuario en entorno Windows

Instalación

Para la instalación, deberemos acceder a las *releases* del Repositorio de la GUI, y descargar el fichero comprimido en `zip` que contiene los binarios de la instalación. También se pueden ver los videos que documentan la instalación de la interfaz de usuario en Windows, disponibles en el Apéndice G.2.

Una vez hayamos descargado los binarios, y tengamos el fichero `i386-engine.dll` en la misma carpeta, podremos ejecutar el simulador.

Ejecución

La ejecución es bastante intuitiva, con botones que se llaman igual que en la variante de Linux y que hacen referencia a los mismos casos de uso (*Step*, *Breakpoint*, *Continue*...). Sin embargo, se pueden encontrar manuales explicativos en la barra superior del menú, llamada "Ayuda". De igual manera, está explicado con detalle en el video disponible en el Apéndice G.2.

Apéndice F

Anexo de sostenibilización curricular

F.1. Introducción

Al comienzo del trabajo, la percepción de sostenibilidad del alumno se reducía únicamente a la sostenibilidad ambiental. Posteriormente, durante el desarrollo el trabajo, dicha concepción ha ido evolucionando, dotando al alumno de una percepción distinta acerca de lo que realmente significa "sostenibilidad".

F.2. Sostenibilidad económica

La sostenibilidad económica responde a la capacidad de mantener el proyecto a lo largo del tiempo. Como nuestro proyecto no requiere pagos periódicos ni de licencias ni de despliegue, el coste de mantenimiento es prácticamente cero. El mantenimiento del software se plantea como un proyecto Open Source donde diferentes usuarios a lo largo y ancho de internet puedan colaborar, solucionar errores y mejorar poco a poco el código.

Para ello es necesario construir una comunidad de usuarios sólida, que puede conseguirse mediante la consecución de que el programa sea utilizado en las universidades de España, empezando por la Universidad de Burgos.

F.3. Sostenibilidad social

La sostenibilidad social corresponde a la necesidad de este proyecto de generar un producto que sea beneficioso a largo plazo, y equitativo para las personas que empleen el software.

Por ello, se ha desarrollado un software cuyos principales objetivos son la inclusión, el desarrollo de las capacidades de los alumnos, que conlleve un impacto educativo a largo plazo, y que se sostenga sobre pilares basados en la ética y en el uso moral y responsable de software de terceros. También es menester fomentar el pensamiento crítico de los alumnos, mediante el uso de este simulador.

F.4. Sostenibilidad ambiental

La sostenibilidad ambiental consiste en desarrollar productos o procedimientos que sean ecológicamente responsables, es decir, que minimicen el impacto que puedan tener en el medio ambiente y en la biodiversidad. Es por ello que el desarrollo del software de este proyecto ha buscado mantenerse en un marco sostenible y tener un coste asimiliable para los usuarios y el planeta.

Según un estudio que involucra a varias universidades portuguesas [7], C es el lenguaje menor coste energético, y uno de los que menos memoria usa. Esto puede parecer irrelevante de primeras, pero si pensamos en como actualmente los *Data Centers* de inteligencia artificial están utilizando energía sin ningún tipo de preocupación acerca de la cantidad que usan, ya que utilizan lenguajes menos eficientes energéticamente como Python, nuestra elección del lenguaje C cobra más sentido. Dicho estudio puede verse reflejado en la Figura F.1.

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Figura F.1: Eficiencia de distintos lenguajes de programación

Apéndice G

Videos

Junto con toda la documentación asociada al proyecto, ya sea la memoria, este mismo documento de anexos, los manuales y demás, se han incluido una serie de videos explicativos que buscan documentar procesos como el de instalación y compilación, u otros aspectos que pueda resultar confuso una explicación escrita. Para ello, se han grabado videos disponibles en la entrega del Trabajo de Fin de Grado, así como en OneDrive. Dichos vídeos se pueden clasificar en 2 según su entorno, ya sea Windows o Linux.

G.1. Linux

Los videos contenidos en esta sección corresponden a la rama del proyecto orientada a Linux, y explican y detallan su instalación, compilación y uso.

Entorno, Instalación, Compilación y Utilización

Este primer video indica como instalar el Simulador, clonando el repositorio de Github, las librerías que deben ser instaladas antes de compilar y como compilar y utilizar el simulador, usando una serie de ejecutables que vienen por defecto en la disrtibución del repositorio.

Podemos encontrar este video como 1-Instalacion-linux.mp4.

Creación de programas de 32 bits

En este segundo video podemos aprender a compilar programas para ejecutarlos con el simulador. Para ello, tenemos 3 opciones.

- Compilar programas en C (deberemos compilar primero nuestro *Cross-Compiler*).
- Compilar programas ASM (deberemos utilizar una herramienta de terceros para compilarlo).
- Escribir programas ejecutables a mano (escribiendo el *bytecode* de cada instrucción).

Podemos encontrar este video como 2-Compilación-programas.mp4.

Debugging

En este tercer video podemos ver de manera práctica y visual la técnica seguida para debuggear el comportamiento del simulador.

Podemos encontrar este video como 3-Journal-Debugging.mp4.

Demostraciones de ciberseguridad

De manera adicional, se han desarrollado una serie de videos que tratan de documentar y expplicar la explotación de varios programas vulnerables mediante la utilización del simulador. Estos videos explican tanto el programa vulnerable, las debilidades o vulnerabilidades presentes, y su posterior explotación e impacto.

Stack Buffer Overflow

Explica la vulnerabilidad *Stack Buffer Overflow* y como explotarla, usando tanto la ejecución de instrucciones en pila (NX desactivado) como la utilización de gadgets para una explotación de ROP (*Return Oriented Programming*) mediante la técnica de *Return to Libc* (Ret2libc).

Podemos encontrar este video como 4-StackBufferoverflow.mp4

Race Condition

Explica la vulnerabilidad *Time-of-Check Time-of-Use* (TOCTOU), también conocida como condición de carrera. También enseña como aprovechar el simulador para su explotación.

Podemos encontrar este video como 5-RaceCondition.mp4

Externally-Controlled Format String

Explica la vulnerabilidad *Externally-Controlled Format String* y muestra como aprovechar el simulador para su explotación a través de la construcción de un *payload* que modifique la dirección de retorno de una función.

Podemos encontrar este video como 6-FormatString.mp4.

Programas en ensamblador

De manera adicional, se han desarrollado una serie de programas en ensamblador que sirven como ejemplo de una programación del mundo real a bajo nivel. Los programas incluyen unas implementaciones personalizadas de `memcpy`, `strlen`, y del algoritmo de suma por desplazamiento.

Podemos encontrar este video como 7-EjemplosASM.mp4

Entorno Docker para la utilización del simulador

Muestra y explica los repositorios adicionales del proyecto, que son los siguientes:

- Toolchain estática del i386 para poder emplearla en el entorno Docker sin tener que compilarla cada vez.
- Entorno Docker para la utilización del Simulador en laboratorio. Permite correr una máquina basada en Ubuntu que tenga instalado tanto el simulador como la toolchain, para que los alumnos puedan conectarse via SSH desde su máquina, siempre que estén en la misma red.

Adicionalmente, muestra un ejemplo de ejecución, pudiendo acceder a la máquina contenerizada desde una segunda máquina en la misma LAN.

Podemos encontrar este video como 11-Ejecución-Entorno-Docker.mp4

G.2. Windows

Los videos contenidos en esta sección corresponden a la rama del proyecto orientada a Windows, y explican y detallan su instalación, compilación y uso.

Compilación de la librería DLL

Muestra el proceso de compilación de la librería DLL empleada por el programa que implementa la GUI del simulador en Windows y las herramientas necesarias para dicha compilación.

Podemos encontrar este vídeo como 8-CompilacionDLL-Windows.mp4

Instalación y Utilización del simulador

Muestra como descargar los binarios de la GUI del simulador para Windows, inyectar el DLL compilado anteriormente, y el modo de empleo de dicha Interfaz Gráfica de Usuario, y como ejecutar programas tanto binarios como ASM.

Podemos encontrar este video como 9-GUI-Simulador-Windows.mp4

Código Fuente GUI

Muestra la organización general y explica el código fuente de la GUI hecha mediante Windows Forms, y como interactúa con la librería DLL para poder ser funcional.

Podemos encontrar este vídeo como 10-CodigoFuenteGUI.mp4

Bibliografía

[1] A. Alcalde. (2026) Cómo crear diagramas de gantt en latex. Accedido: 6 de abril de 2026. [Online]. Available: https://elbauldelprogramador.com/crear-diagrama-de-gantt-en-latex/

[2] S. Boutnaru. (2022) Security — nx bit (non-executable). Accedido: 25 de abril de 2026. [Online]. Available: https://medium.com/@boutnaru/security-nx-bit-non-executable-18759fd2802e

[3] E. University. (2025) Qué es el modelo-vista-controlador, cómo funciona y ventajas. Accedido: 25 de abril de 2026. [Online]. Available: https://www.esic.edu/rethink/tecnologia/modelo-vista-controlador-que-es-como-funciona-y-ventajas-c

[4] Massachusetts Institute of Technology, "Intel 80386 reference programmer's manual: Table of contents," 2007, accedido: 25 de febrero de 2026. [Online]. Available: https://pdos.csail.mit.edu/6.828/2007/readings/i386/toc.htm

[5] Linux man-pages project, *vdso(7) — overview of the virtual ELF dynamic shared object*, Linux Kernel Documentation, 2024, accedido: 25 de abril de 2026. [Online]. Available: https://man7.org/linux/man-pages/man7/vdso.7.html

[6] NASM Development Team, "Nasm - the netwide assembler," 2025, versión 2.x, Accedido: 27 de marzo de 2026. [Online]. Available: https://www.nasm.us/

[7] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes, and J. a. Saraiva, "Energy efficiency across programming languages:

how do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 256–267. [Online]. Available: https://doi.org/10.1145/3136014.3136031