UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

TFG del Grado en Ingeniería Informática

Simulador del microprocesador 80386

Un enfoque orientado a docencia

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

Agradecimientos

En primer lugar, quiero agradecer a mi padre por tantos años de apoyo incondicional, de consejos de valor incalculable y sobre todo, de paciencia, que me han hecho llegar a donde estoy y convertirme en la persona que soy hoy día, tanto profesional como personalmente. Agradecer también, al resto de mi familia, a mi madre, mi hermana, mi abuela, y en especial a mi abuelo, por las oportunidades que me han brindado, por la comprensión y por todo lo que han hecho por mí.

También quisiera agradecer a mi tutor, el profesor Pedro Sánchez Ortega, por su apoyo y confianza en este proyecto. Por formar parte de este Trabajo de Fin de Grado y ayudarme a darle forma de la manera que inicialmente pensé. Sin su colaboración, este proyecto no sería lo que es.

No me puedo olvidar de mis amigos, la familia que se elige, y que tantas veces han tenido que oírme hablar de este proyecto, seguramente sin entender ni a qué me refería. A ellos, en especial a Víctor, Juan Carlos, Antonio, Guillermo, Diego Arbeloa y Diego Guerrero, tengo que agradecerles también el haberme amenizado y llenado de risas y recuerdos estos últimos cuatro años. Por otro lado, a Rubén, Silvia, Estela y el resto de "Polayers" también quiero agradecerles el tiempo que compartimos, los consejos que intercambiamos, las risas, las bromas y los momentos juntos. Gracias también, a Gabriela, por ser un pilar fundamental durante estos últimos meses, escucharme cuando lo he necesitado y por apoyarme y animarme durante todo este tiempo.

A Patricia, por ser la persona que necesitaba, que me acompañe en todo y con la que compartir lo bueno y lo malo. Gracias por ser como eres y por creer en mí, por aguantarme con mis más y mis menos, pero sobre todo, por estar ahí.

Si hay un agradecimiento casi anterior a la concepción de este proyecto, es al Dr. Héctor Marco Gisbert, de la Universidad Politécnica de Valencia, por iniciarme en el mundo del Hacking Ético e inspirar este proyecto.

En general, muchas gracias a todos los que han creído en mí, a los que he querido y me han querido con mis defectos y mis virtudes, y con los que he compartido un trozo del camino durante estos últimos cuatro años. Mil gracias por estar.

Resumen

El objetivo de este trabajo es la concepción y posterior desarrollo de un simulador del microprocesador 80386 desarrollado por Intel. Dicho procesador, también conocido como i386, es un procesador de 32 bits. La finalidad del simulador reside en ser utilizado para docencia en varias asignaturas de la titulación de Ingeniería Informática de la Universidad de Burgos u otras universidades. Algunos ejemplos de estas asignaturas podrían ser Fundamentos de Computadores o Ciberseguridad.

La idea principal del simulador, es ofrecer una interfaz clara y sencilla al usuario, que permita seguir y comprender la ejecución de programas en ensamblador paso a paso, tanto para aprender cómo funcionan los microprocesadores y su programación, como para entender la explotación de programas vulnerables con CWE conocidos. Esto se consigue a través de la consecución de que el simulador pueda ejecutar ficheros ejecutables compilados para el i386 (o bien ELF32 o bien *bytecode* ejecutable sin cabeceras).

Esta herramienta está concebida para ser ejecutada en un entorno Linux, a través de la línea de comandos. Este entorno resulta ideal por su facilidad para escribir *scripts*, compilar ejecutables, efectuar redirecciones a descriptores de ficheros, la utilización de tuberías, desensamblar programas en hexadecimal, y sobre todo, para la ejecución de *syscalls* o llamadas al sistema de Linux usando la ABI del i386.

De manera adicional, se ha desarrollado, como otra rama del mismo proyecto, una interfaz gráfica que permite ejecutar el simulador desde el sistema operativo Windows. Esta rama está orientada a la docencia de Fundamentos de Computadores, ya que algunas funcionalidades avanzadas han sido recortadas por su incompatibilidad en Windows.

Para ello, el software se implementa como una librería multiplataforma que pueda ser compilada y utilizada de igual manera en distintos sistemas operativos, donde lo único que varía es la interfaz gráfica o la manera en la que se compila el programa final.

Descriptores

Simulador, Ensamblador, i386, Ciberseguridad, Compilador, Llamadas al sistema, ABI, Compilación Estática, ELF-32, CWE, Exploits, 32-bit, Docencia.

Abstract

The aim of this project is the conception and subsequent development of an Intel 80386 microprocessor simulator. This microprocessor, widely known as the i386, is a 32-bit CPU. The main goal of the simulator is to serve as a teaching resource for several subjects within the Computer Engineering Degree at the University of Burgos (or any other university), including Cybersecurity and Computer Fundamentals.

The main idea of the simulator is to provide a clean and intuitive interface to the user, that allows them to follow and comprehend the execution of assembly programs step by step. This makes it possible to learn how microprocessors work (and their programming) while also demonstrating how vulnerable programs, with known CWEs, can be exploited. This is achieved by enabling the simulator to execute both ELF32 and assembly programs.

This tool is designed to be run in a Linux environment, through the command line. Such environment is particularly suitable due to its capacity to run scripts, compile object files, use pipes and redirections, disassemble programs to hexadecimal code, and above all, to execute system calls using the Linux i386 ABI.

In addition, a Graphical User Interface (GUI) has been developed as another branch of the same project, allowing the simulator to be executed under the Windows operating system. This branch is mainly focused on teaching Computer Fundamentals, since some advanced functionalities have been removed due to their incompatibility with Windows.

To achieve this, the software has been implemented as a cross-platform library that can be compiled and used in the same way across different operating systems, where only the graphical interface or the method used to compile the final program varies.

Keywords

Simulator, Assembly, i386, Cybersecurity, Compiler, System calls, ABI, Static Compiling, ELF-32, CWE, Exploits, Teaching.

Índice general

Índice de figuras

Índice de tablas

1. Introducción

Al iniciarse en el mundo de la informática, el funcionamiento de un microprocesador, y por ello el de gran parte de un ordenador puede resultar confuso y muy difícil de entender para los nuevos estudiantes. Conceptos como *instrucciones*, *dirección de memoria*, *memoria virtual*, *puntero* o *pila* son a menudo dados por entendidos y pueden llegar a formar una "bola de nieve" que retrasa el aprendizaje.

Asimismo, es contra-intuitivo pensar que un procesador puede llegar a hacer prácticamente cualquier cosa cuando lo único que vemos es que mueve datos de un registro a otro, modifica direcciones de memoria, o realiza operaciones aritméticas sencillas entre dos operandos. Por otro lado, los estudiantes tienden a no estar familiarizados con el concepto de "arquitectura", cuáles son las distintas arquitecturas que existen, en qué difiere el número de bits de ésta, y en qué se diferencian entre ellas.

En consecuencia, a la hora de aprender el lenguaje C (muy trabajado al inicio de la titulación), el aprendizaje y comprensión de estos conceptos y otros relacionados con la naturaleza del lenguaje, como el *preprocesador* pueden llegar a hacerse cuesta arriba. Con una base sólida, los alumnos serán capaces de entender qué ocurre sobre la máquina física cuando ejecutan un programa cualquiera, e incluso desarrollar un nivel alto en C, que resulta muy útil cuando buscamos que el rendimiento de nuestro programa sea óptimo, ya que C es de los lenguajes con una ejecución más rápida.

Como respuesta a estas problemáticas, nace la idea de desarrollar un simulador. Según la RAE [1], un "simulador" es un aparato que reproduce artificialmente el funcionamiento real de un sistema, y que se usa generalmente para entrenar a quien deben manejar ese sistema. Dicha simulación

se puede afrontar de varias maneras, ya sea de manera operativa (que se comporte igual) o de manera electrónica (que sea igual). En este trabajo se va a optar por una simulación operativa, pero buscando la mayor fidelidad al microprocesador Intel 80386 real.

La idea de que dicho simulador correspondiera al i386 reside en una serie de características de éste. En primer lugar, el i386 es el primer microprocesador capaz de ejecutar Linux, lo cual lo vuelve una potente alternativa para iniciarse en el mundo del bajo nivel.

En segundo lugar, es un microprocesador de 32 bits. Esto significa que tiene más posibilidades que un microprocesador de 16 bits (el impartido en docencia hasta la fecha, mediante el uso del Simulador del 8085 de la Universidad de Granada [2]), pero sin llegar a la complejidad que pueda entrañar un microprocesador de 64 bits. De manera adicional, las direcciones de memoria de 32 bits son más visuales y más fáciles de recordar. Además, la mayoría de máquinas actuales, que tienden a ser en su totalidad de 64 bits, tienen un modo de compatibilidad de 32 bits, lo que les permite ejecutar programas de 32 bits.

Por último, el número de instrucciones del Intel 80386 no es muy elevado, y sus *nemotécnicos* tienden a ser bastante auto-explicativos. Con esto, y un manual detallado que especifique cómo han de programarse ejecutables para este microprocesador, podemos conseguir un entorno amigable de cara a la docencia universitaria.

Si establecemos el enfoque en la ciberseguridad, con sólo un par de instrucciones, establecidas estratégicamente a nuestro gusto en un punto crítico, podemos llegar a hacer desde una *Denegación de Servicio (DoS)*, hasta tomar el control por completo de la máquina, incluso con privilegios de administrador. De igual manera, estos objetivos pueden llegar a alcanzarse sin ni siquiera inyectar código, tan sólo sabiendo cómo funciona un microprocesador, y aprovechando un programa vulnerable.

Existen multitud de técnicas de explotación que pueden llevarse a cabo en un entorno de bajo nivel, rozando el nivel de ensamblador, y en este trabajo se abordarán algunas de ellas como prueba de que, con un conocimiento amplio sobre ensamblador, el lenguaje C, arquitecturas... etc, pueden conseguirse explotaciones absurdamente efectivas. Este simulador también nace como una herramienta que haga más sencilla la comprensión

del funcionamiento de estas técnicas, y cómo funcionan los *exploits* que las implementan. Para este proyecto, se entregan los siguientes materiales.

- Repositorio del simulador, tanto su rama main como su rama win32. La rama `main` está desarrollada para Linux, mientras que la rama `win32` está desarrollada para ser compilada como DLL y empleada como motor del simulador en Windows, junto con la GUI que se ha desarrollado.
- Repositorio de la Interfaz gráfica para Windows.
- Repositorio del Entorno docker para la ejecución del simulador
- Repositorio de la Toolchain estática del i386
- Binarios compilados para Windows.
- Vídeos explicativos, incluyendo un índice que explica qué contiene cada video.
- Manuales de las pruebas de concepto de ciberseguridad.
- Memoria del proyecto.
- Anexos.
- Presentación PowerPoint.

2. Objetivos del proyecto

El objetivo de este trabajo es familiarizar a los estudiantes con el Intel 80386, para que puedan aprender cómo funcionan los computadores a nivel de ensamblador, y ayudar a los docentes con dicha tarea. Además, también se busca enseñar a los estudiantes cómo afectan las vulnerabilidades de un programa a nivel de ensamblador y cómo estas pueden explotarse, pudiendo controlar el flujo del programa, la pila y los valores de los registros de la CPU. El simulador se concibe para que estas demostraciones de ataques a bajo nivel sean más sencillas de comprender y de replicar, ya que, por ejemplo, en un *Stack Buffer Overflow Attack*, puede ser útil ver la pila y cómo cambia instrucción a instrucción.

2.1. Objetivos basados en requisitos funcionales

Los objetivos que debe satisfacer el programa, en el sentido de las funcionalidades que debe implementar, son los siguientes:

- El simulador debe permitir dos modos de ejecución, uno con interfaz gráfica integrada en la línea de comandos, y otro desde la propia línea de comandos, de manera que sea posible efectuar redirecciones mediante tuberías. En cuanto a la interfaz gráfica, debe contar con varias ventanas o zonas. Una en la que sean visibles los registros de la CPU, otra en la que sean visibles las instrucciones del código, otra en la que sean visibles los valores más bajos de la pila, y otra que funcione como terminal para las operaciones que permite el simulador,

como poner *breakpoints* o consultar el contenido de ciertas direcciones de memoria.

- El simulador debe poder ejecutar el juego de instrucciones del Intel 80386. Para ello, los ejecutables deben estar compilados para el i386, y no contener instrucciones ajenas a éste. El simulador debe poder ejecutar ficheros ejecutables compilados estáticamente y con No-PIE, es decir, siempre en las mismas direcciones de memoria y con todas las librerías necesarias incorporadas en el ejecutable. Esto tiene como objetivo evitar el enlazamiento dinámico de librerías, por su complejidad. La ejecución de programas será monohilo.

- El simulador debe poder ejecutar tanto ficheros ejecutables en formato ELF-32 para i386 con las características anteriormente mencionadas (compilados estáticamente y sin PIE) como ficheros sin cabeceras. Estos ficheros sin cabeceras, de ahora en adelante ficheros *Raw*, son ficheros que, desde su primer byte, hasta su último byte, contienen *bytecode* traducible a instrucciones del 80386. Estos ficheros pueden crearse en herramientas como **hexedit**, en el entorno de Linux.

- La implementación de las estructuras de datos del microprocesador (registros, memoria, pila) debe ser completa, a excepción de los puertos de entrada y salida, cuyo uso es prácticamente nulo en programas sencillos como los que se van a usar en el simulador.

- La implementación de las llamadas al sistema (o *syscalls*) será parcial, permitiendo ejecutar programas sencillos, como aquellos que leen un archivo, o leen y escriben por la terminal. Dichas llamadas al sistema inicialmente tendrán soporte para la arquitectura x86_64 (la más extendida), pudiendo añadir soporte para ARM (arm64) más adelante.

2.2. Objetivos técnicos

Los siguientes objetivos están relacionados con los aspectos técnicos que se esperan del simulador.

- El simulador debe poder ejecutarse en local, en una máquina con arquitectura x86_64 (también llamada amd64) de 64 bits. De cara a la docencia, para evitar la instalación, el simulador podrá estar alojado en un contenedor[1] con la arquitectura anteriormente mencionada

[1] Docker con la imagen de Ubuntu-22.04

y que permita a los alumnos acceder mediante protocolo SSH[2] a dicho contenedor. Dicho contenedor debe crear las carpetas y usuarios necesarios para ello.

- El simulador debe tener un rendimiento aceptable, ejecutando cada instrucción de manera que el usuario la perciba como instantánea.

2.3. Objetivos personales

A continuación se va a proceder a exponer una serie de objetivos del alumno en el marco de este proyecto.

- Introducir líneas de docencia provenientes de otras instituciones[3] en campos poco trabajados, como puede ser la ciberseguridad a bajo nivel.

- Conseguir que el simulador sea lo suficientemente completo como para ser utilizado en la Universidad de Burgos, e incluso en otras universidades, para que los alumnos puedan comprender ensamblador y la explotación de programas a bajo nivel de una manera más sencilla, de manera que lleguen a desarrollar interés por este campo de conocimiento. En el contexto de la Universidad de Burgos, que pueda utilizarse en la titulación de Ingeniería Informática, en las asignaturas de Fundamentos de Computadores y Seguridad Informática.

- Permitir a otros alumnos desarrollar un nivel alto en el lenguaje C. Basado en la premisa de que es un lenguaje muy potente, altamente ligado a la programación de sistemas operativos, y de muy alto rendimiento. El desarrollo del proyecto ha resultado en un incremento del nivel del alumno en C, desarrollando incluso una afinidad con éste, algo que bajo su criterio, no abunda en la titulación. El simulador va parcialmente unido a la ejecución de programas sencillos compilados en C, para luego depurarlos en el simulador, y realizar ataques sobre las vulnerabilidades que puedan tener estos. Para realizar dichos ataques hace falta conocer como funciona C, los punteros, la reserva de memoria, y las principales vulnerabilidades, consiguiendo así una experiencia que puede resultar enriquecedora para los alumnos.

[2]Secure Shell, permite el acceso por terminal remota a un equipo

[3]Basado parcialmente en la experiencia del alumno en la Universidad Politécnica de Valencia

- Abrir la puerta a que otros alumnos puedan desarrollar proyectos en el marco teórico de éste, como podrían ser un compilador para el i386 de instrucciones escritas en ensamblador, y lo transforme en bytecode que entienda el i386, o un simulador de las versiones superiores de los microprocesadores de Intel, como el 80486 o el 80686.

- Por último, uno de los objetivos personales más relevantes de todo el proyecto, era llevarlo a cabo como reto personal e intelectual. Es decir, para que el alumno pudiera demostrarse a sí mismo, y a la comunidad universitaria, que con sus conocimientos actuales era capaz de desarrollar algo tan complejo como este simulador. Este proyecto además de buscar desarrollar un software útil y de calidad para docencia, también buscaba mejorar las *skills* y los conocimientos del alumno.

3. Conceptos teóricos

Con la finalidad de permitir una mejor comprensión de las tareas realizadas en el presente proyecto, de los conceptos explicados en este documento, y las aplicaciones para docencia que puede llegar a tener el simulador, se van a proveer una serie de definiciones y aclaraciones.

3.1. Conceptos relativos al microprocesador

Instrucción unidad mínima ejecutable que puede correr un microprocesador. Realiza una operación específica sobre el *hardware* del microprocesador.

Nemotécnico abreviatura del nombre de una instrucción. Identifica de forma única la instrucción.

Registro estructura interna del microprocesador que es capaz de almacenar datos. En el caso del Intel 80386, los datos pueden ser de hasta 32 bits. Cada registro puede ser de propósito general o tener un propósito específico.

Arquitectura estructura y diseño de los componentes internos del procesador. Determina qué operaciones puede realizar y con ello, las instrucciones que el procesador será capaz de ejecutar.

Pila estructura de datos que reside en la memoria principal del microprocesador. Su trabajo consiste en almacenar datos, de manera que el último dato que ha entrado en la pila será el primero en salir, siguiendo un paradigma *LIFO*.

Rutina también llamado función. Conjunto autónomo de instrucciones que realiza una tarea específica y que puede ser invocado repetidamente, tomando argumentos o no.

Byte unidad fundamental de almacenamiento que está compuesto de 8 bits, que son valores binarios, 1's o 0's.

Direccionamiento modo en el que se obtienen los argumentos para una instrucción.

Bytecode conjunto de bytes que corresponden a una serie de instrucciones interpretables por el microprocesador. Es el código ejecutable que entiende el procesador.

Marco de pila conjunto de bytes reservados dentro de la pila para una determinada función.

Base frame pointer puntero a una dirección fija dentro del marco de pila que permite acceder de una manera estable a los argumentos y variables de la función, independientemente del cambio de puntero a la pila (ESP). El base frame pointer suele corresponder al registro EBP.

Puntero variable que almacena la dirección de memoria de otra variable o dato.

Dirección de memoria identificador que hace referencia a un único byte en la memoria principal.

Espacio de direcciones conjunto de direcciones de memoria que son accesibles en un equipo. En el caso del i386, va desde 0x00000000 hasta 0xFFFFFFFF.

Memoria física soporte físico sobre el que se almacena la información.

Memoria virtual técnica mediante la cual los procesos utilizan direcciones de memoria virtuales, que son traducidas por el MMU [3] y convertidas en direcciones de memoria físicas.

Segmentación técnica mediante la cual la memoria del procesador es divida en segmentos para aislar bloques de memoria, u ofrecer protección a determinados rangos de direcciones.

Modo *Flat* modo de ejecución en el que solo existe un solo segmento, es decir, todos los segmentos son ignorados.

Modo protegido base de muchos sistemas operativos modernos. En el i386 el espacio de direcciones es de 32 bits (4GB).

Modo real modo de arranque del microprocesador i386. En el i386 permite direcciones de hasta 20 bits (1MB).

Modo VM 8086 modo de compatibilidad con el procesador 8086.

Interrupción evento en el cual se interrumpe la ejecución en el microprocesador y se ejecuta una rutina específica en función del número de interrupción.

3.2. Conceptos relativos al sistema operativo

Loader componente del sistema operativo que carga programas en memoria.

No-PIE (No-Position Independent Executable) ejecutable que debe ser cargado siempre en las mismas direcciones de memoria.

ELF (*Executable and Linkable Format*) es un formato de archivo binario estándar para ejecutables y librerías [4].

Program Headers estructura interna del ELF que define los segmentos de memoria necesarios para ejecutar el programa, por lo que deben ser cargados en memoria.

Section Headers estructura interna del ELF que define el contenido y la ubicación de las distintas secciones del archivo. Por ejemplo, cuáles son código ejecutable y cuáles están reservadas para almacenar datos.

Descriptor de fichero identificador numérico asignado a cada recurso por el sistema operativo. En Linux, la entrada estándar es el 0, la salida estándar el 1, y la salida de errores el 2.

Linker componente del sistema operativo que enlaza el fichero resultante de la compilación (fichero objeto) con bibliotecas para obtener el fichero ejecutable. Si este enlace se produce en tiempo de ejecución, se denomina “Enlazamiento dinámico”. En el presente proyecto se ha evitado el enlazamiento dinámico mediante la compilación estática, que añade al ejecutable todas las librerías necesarias en tiempo de compilación.

Syscall mecanismo que permite solicitar un servicio del sistema operativo, cediendo el control de ejecución temporalmente al *kernel* [5].

Kernel componente fundamental del sistema operativo, actúa como intermediario entre el *hardware* y el *software*. También se le conoce como núcleo.

ABI (*Application Binary Interface*) especificaciones a bajo nivel que indica como deben interactuar los programas y el sistema operativo a nivel de código máquina. Por ejemplo, si los argumentos para una función o una llamada al sistema van en los registros (y en qué orden) o en la pila.

API (*Application Programming Interface*) conjunto de reglas y especificaciones que permiten interactuar a dos o más *softwares*.

Variables de entorno permiten definir variables críticas fuera de los programas, en el entorno del sistema operativo, como pueden ser el PATH o la contraseña para una aplicación.

Secuencia de escape secuencia de caracteres que permite realizar una función específica, como cambiar el color del texto, limpiar la salida estándar, o mover el puntero de ésta.

Tuberías transforma la salida estándar de un comando en la entrada estándar del siguiente.

Redirecciones envía la salida estándar de un comando a un fichero o descriptor de fichero.

3.3. Conceptos relativos al lenguaje C

Compilador programa que traduce un código fuente junto con sus librerías a un fichero objeto.

Cross Compiler compilador capaz de compilar un programa para una arquitectura distinta a sobre la que está ejecutándose él. Por ejemplo, un compilador capaz de compilar un programa de amd64 estando en un sistema operativo arm64.

Compilación estática técnica que permite enlazar todas las librerías necesarias en tiempo de compilación para evitar tener que enlazarlas en tiempo de ejecución.

Tipos de datos distintas formas de clasificar la información. En C, prácticamente todos los tipos simples son enteros, pero lo que cambia es su forma de manipularlos.

Fichero cabecera fichero que contiene declaraciones y prototipos de funciones, macros, y otros elementos para que varios programas puedan compartir información entre ellos.

Fichero fuente fichero que contiene el código fuente del programa. Puede utilizar ficheros cabecera o no.

Preprocesador programa que analiza el código fuente y lo prepara para su posterior compilación.

Macros conjunto de instrucciones que el preprocesador reemplaza en el código fuente antes de la compilación.

3.4. Conceptos relativos a la ciberseguridad

Técnicas de mitigación técnicas que permiten evitar (o mitigar) algunos de los ataques de explotación más comunes.

Prueba de concepto demostración del concepto a pequeña escala que busca verificar la viabilidad de un proceso. En el contexto de este documento, las pruebas de concepto (PoC) buscarán demostrar como un ataque de explotación puede ser efectivo y replicable.

Ataque de explotación (Exploit) ataque que busca explotar una vulnerabilidad conocida con la finalidad de alterar el flujo del programa a su gusto.

Explotación a bajo nivel técnica que busca explotar un programa atacando la ejecución a bajo nivel de éste, intentando modificar la memoria o las estructuras de datos del microprocesador.

CWE (*Common Weakness Enumeration*) [6] lista de debilidades conocidas de *software* y *hardware* que pueden desembocar en vulnerabilidades que puedan ser atacadas.

NX (*Non-Executable*) técnica de mitigación que impide que la información de la pila sea ejecutable, saltando una excepción y terminando el programa si se intenta ejecutar una instrucción proveniente de la pila.

ASLR (*Address Space Layout Randomization*) técnica de mitigación que busca aleatorizar las direcciones de memoria del programa, impidiendo así la replicación de ciertos ataques de explotación.

SSP (*Stack Smashing Protector*) técnica de mitigación que consiste en establecer un valor conocido en la pila cada vez que el programa requiera que el usuario escriba sobre la pila. De esta manera, si el usuario sobrepasa la cantidad de valores que puede escribir, y sobrescribe este valor (llamado comúnmente canario), al realizar la comprobación y comprobar que el valor ha cambiado, se lanza una excepción y se interrumpe el programa.

Debilidad fallo en la programación que desemboca en un comportamiento indeseado o deja abierta la puerta a un malfuncionamiento.

Vulnerabilidad existe una vulnerabilidad si una debilidad puede ser atacada y aprovechada por un atacante en su beneficio.

Stack Buffer Overflow (CWE-121) debilidad que consiste en la posibilidad de escribir en un buffer (que resida en la pila) más allá de lo esperado por el programa. De esta manera, se puede sobrescribir la pila, por lo que se puede alterar el flujo del programa.

Race Condition (CWE-362) debilidad que consiste en el uso de un recurso compartido sin la correspondiente sincronización.

Ret2lib técnica de explotación que utiliza el Stack Buffer Overflow, y permite tomar el control del flujo del programa, saltando a los finales de las rutinas, que suelen sacar valores de la pila a los registros (instrucción POP), y salir de la rutina (instrucción RET). Al salir de la rutina, se saca el primer valor de la pila y se establece como nuevo EIP (Instruction Pointer). Encadenando varias funciones, siempre y cuando se tenga el control de los valores de la pila, se pueden controlar los valores de los registros y el flujo de ejecución del código.

Gadget respectivo a la anterior técnica de Ret2lib, un gadget es una instrucción o conjunto de instrucciones que corresponden a librerías del sistema, que pueden utilizarse parcialmente para tomar el control del programa.

Payload secuencia de bytes que aprovechan una vulnerabilidad y permiten realizar la explotación de un programa.

Shellcode payload que consigue una terminal de comandos en un programa que no estaba pensado para ello.

Externally-Controlled Format String debilidad que consiste en dejar una cadena de formato bajo el control del usuario, y que permite leer y escribir valores en memoria de manera no deseada.

4. Técnicas y herramientas

En el presente capítulo se van a exponer las diferentes herramientas, librerías, y técnicas que han sido utilizadas a la hora del desarrollo del software del simulador, y las razones que ha habido detrás de dichas elecciones.

4.1. Lenguaje de programación C

La elección de utilizar el lenguaje C para el desarrollo del simulador está basado en ciertas características intrínsecas al propio lenguaje. En primer lugar, C es un lenguaje de ejecución muy rápida, y compilado, que tienden a ser más rápidos que los interpretados. La compilación además permite un alto nivel de optimización, y personalizar parámetros de la compilación. La personalización de los parámetros de compilación también nos va a ser útil, ya que nos va a permitir compilar programas estáticamente y con No-PIE.

En segundo lugar, C está muy próximo al lenguaje ensamblador, de manera que prácticamente podríamos traducir lo que hace cada línea de código a ensamblador con solo un vistazo. Además, la librería estándar de C (o *libc*) incluye *wrappers* para las llamadas al sistema, que resultan fundamentales para las demostraciones de ciberseguridad de explotación a bajo nivel. Esto nos permite poder ejecutar llamadas al sistema directamente desde nuestro código, y eligiendo los argumentos que queremos pasarle, cosa que suele ser opaca al programador en otros lenguajes.

En tercer lugar, y siendo el motivo principal de la elección de este lenguaje para el desarrollo, tenemos la capacidad para gestionar la memoria. C funciona de tal manera que es el usuario el que debe reservar manualmente la memoria para utilizar, y liberarla cuando ya no la vaya a usar. El 80386 tiene un espacio de direcciones de 2^{32} bytes. Esto significa que para la

ejecución del simulador necesitamos reservar aproximadamente 4 GB de memoria principal para la ejecución. Estos 4 GB no se reservan de manera que se consuman de la memoria principal y ya no se puedan usar, lo que se hace es reservarle un espacio de direcciones virtuales de 4.294.967.296 bytes (4 GB), pero la memoria real usada solo es aquella a la que se accede en el programa, que no suele llegar a 3 MB. Esto se debe a que hasta que las direcciones virtuales no son accedidas para lectura o escritura, el MMU no les asigna una dirección física.

Por último, la utilización de C sería a la vez un reto intelectual (por la complejidad que presenta para usuarios primerizos) y la manera más eficiente de llevar a cabo el simulador, ya que todas las funcionalidades implementadas serían, al menos a juicio del alumno, más enrevesadas de implementar en otros lenguajes como Java o Python.

Para la compilación del simulador, se ha utilizado `gcc`, mediante un *script* en `bash` (disponible en el repositorio) que se encarga de compilar todo el simulador sin tener que escribir todas las opciones cada vez.

4.2. Librerías utilizadas

Para el desarrollo del software, se contempló la posibilidad de utilizar dos librerías. Una para desensamblaje y otra para la interfaz gráfica.

Desensamblaje

Al inicio del desarrollo, la idea principal fue hacer el desensamblaje a mano, byte a byte, pero esto era una tarea ardua ya que una misma instrucción puede tener distintos *opcodes*, que viene a ser el byte que corresponde a esa instrucción, y en función de sus argumentos, la instrucción también puede tener un tamaño en bytes variable. Por ello, se contempló la utilización de la librería Capstone.

La librería Capstone permite decodificar un array de bytes en un array del `struct cs_insn`. Este *struct* contiene toda la información necesaria, el nemotécnico, el tamaño de la instrucción, los argumentos, el opcode... De esta manera, es mucho más sencillo desensamblar las instrucciones, y extraer los argumentos, lo que se traduce como una mayor parametrización del código, y en un menor número de líneas para una misma tarea. Sin embargo, hay que tener cuidado a la hora del desensamblaje, ya que se han de configurar una serie de parámetros. Además, el desensamblaje corresponde a la sintaxis

de Intel, y no a la de AT&T, que es la que algunas herramientas usan por defecto. En el producto final, está integrada esta librería en el código.

Interfaz gráfica

Para el desarrollo de la interfaz gráfica integrada en la terminal de Linux, inicialmente se contempló la posibilidad del uso de la librería Ncurses.

Esta librería permite mostrar varias ventanas distintas, con la posibilidad de tamaño variable, y con una muy buena tasa de actualización de pantalla. Sin embargo, esta librería funciona muy bien si el programa solo usa la interfaz, pero si además de la interfaz se quiere que de vez en cuando el programa muestre algo por la salida estándar, resulta un problema, ya que consume la pantalla entera y no deja ver la salida estándar.

Por ello, para la ejecución de las demostraciones de concepto de ciberseguridad, algunos programas que utilizaban las llamadas al sistema de *read* o *write* o alguna de sus variantes, quedaban inutilizados. En consecuencia de esta problemática, se optó por hacer una interfaz gráfica personalizada a mano, sin la librería Ncurses. Esto se consiguió mediante la utilización de secuencias de escape para imprimir colores, caracteres personalizados para imprimir las ventanas, borrar líneas, o para mover el puntero sobre la terminal. En el producto final, no se utiliza la librería Ncurses.

Para la interfaz gráfica del ejecutable de simulador para Windows, se ha utilizado el paquete de herramientas de Visual Studio, junto con las librerías de .NET y Windows Forms, para poder crear la interfaz gráfica de manera rápida y sencilla, posteriormente implementando la lógica de la aplicación con el lenguaje `C#`.

Librería estándar de C

Como sustituto para la librería estándar de C para la compilación de ejecutables i386 se ha utilizado Musl, que es una implementación más liviana de la librería estándar de C, y que resulta suficiente para compilar programas sencillos.

No se debe confundir la librería usada para compilar el simulador, que realmente sí que es la librería estándar de C (*libc* de ahora en adelante), con la librería usada para el *cross compiler* que nos permite generar ejecutables para el i386. Musl nos permite compilar un *cross compiler* y varios programas útiles, como Objdump o Readelf.

Esta elección ha sido motivada por la incapacidad del compilador por defecto `gcc` de compilar programas para i386. Cuando especificamos la opción **-m32** con el compilador `gcc`, el código máquina generado por éste no corresponde sólo al i386, sino que también puede contener instrucciones del i686, por lo que es mejor evitarlo.

Así que, una vez sabemos que debemos compilar de nuevo todos los útiles de programación para poder generar ejecutables i386, mejor utilizar una implementación más sencilla, puesto que el tiempo de compilación es bastante prolongado. De esta manera conseguimos abreviar este proceso.

Una vez construido el *cross compiler*, podemos generar ejecutables que sí que contiene únicamente instrucciones del i386. Además, incluye otros programas como Objdump, que tiende a no funcionar para desensamblar programas x86 en arquitecturas distintas, como ARM.

4.3. Programas auxiliares

Como programas auxiliares en el desarrollo del simulador, se han utilizado ciertos programas del entorno de Linux. Estos programas han ayudado considerablemente al desarrollo de ciertas funcionalidades y en su depuración, ya que, sobre todo en el campo del bajo nivel, un byte erróneo o una *flag* mal puesta puede determinar el flujo entero del programa.

Objdump

Objdump es una herramienta del entorno de Linux que permite desensamblar un ejecutable, de manera que se pueden ver las distintas rutinas, sus nombres, y las diferentes instrucciones que las componen, tanto el opcode de la instrucción como el nemotécnico y sus argumentos.

Objdump muestra por defecto el desensamblaje con la sintaxis AT&T, por lo que si queremos que muestre todo con la sintaxis de Intel, debemos especificar la opción "-Mintel".

Esta herramienta ha sido muy útil para comprobar cómo se debían extraer los argumentos, si el desensamblaje se estaba realizando correctamente, o si una instrucción era muy común en la mayoría de programas o no.

Readelf

Readelf es un programa que permite leer las cabeceras de un fichero ELF. De esta manera se pueden ver los *offsets* donde comienza una sección del

ELF, las tablas de *Program Headers* y *Section Headers* y otras estructuras pertenecientes al fichero ELF. También es una herramienta del entorno de Linux.

Su utilización ha ayudado a la depuración del código fuente encargado de realizar la función de *Loader* en el simulador.

GDB (GNU Debugger)

GDB es una herramienta del entorno de Linux. Se encarga de depurar un ejecutable, permitiendo una ejecución instrucción a instrucción. A diferencia de este simulador, GDB no simula la ejecución, sino que lo ejecuta con el modo de compatibilidad del kernel de Linux (en el caso de los ejecutables de 32 bits), se vincula al proceso (*attach* en inglés), y va mostrando las modificaciones que el programa efectúa en los registros y la pila mediante el uso de señales del kernel, pero no ejecuta el programa en sí mismo. Esto provoca que si el kernel no dispone de compatibilidad con el ejecutable, GDB no funcionará, mientras que el simulador sí.

GDB ha sido usado para depuración, para comprobar si el flujo de código del simulador iba por el mismo camino que el de GDB al ejecutar el mismo ejecutable. De esta manera si los flujos de programa discrepaban, podíamos saber que existía un *bug* en la implementación de alguna instrucción que había que resolver.

Ausyscall

Ausyscall es una herramienta del entorno de Linux que lista todas las llamadas al sistema con su número identificador, dada una arquitectura.

Ha sido utilizada a la hora de traducir los números de llamada al sistema de 32 bits (para el i386) a llamadas al sistema de 64 bits (x86_64). Esto se debe a que los números de llamadas al sistema cambian de una arquitectura a otra. Por ejemplo, *exit()* (que hace que el programa termine) corresponde al identificador 1 en 32 bits, mientras que en 64 bits (x86_64) corresponde al número 60.

NASM

NASM es un programa de terceros que permite compilar programas en ensamblador a distintos formatos. En este proyecto ha sido usado para compilar ficheros en ensamblador a ficheros binarios. Es usado internamente por la GUI para Windows.

4.4. Manuales

Para la tarea de desarrollo del simulador, una fuente confiable de información han sido los manuales. A continuación se va a explicar brevemente los manuales que han sido consultados.

Manual Intel i386 del MIT

El primer manual que ha sido consultado, sobre todo a la hora de entender las estructuras internas de la CPU, la arquitectura interna y comportamiento de la misma, y las operaciones específicas que realiza cada instrucción, es la documentación de la asignatura de *Operating System Engineering* (6828) del Instituto Tecnológico de Massachusetts (MIT)[7].

Dicho manual contiene información esencial sobre los registros, *flags*, interrupciones, instrucciones, segmentación, y los modos de ejecución del i386.

Manual de Linux

El segundo manual que ha sido consultado, ha sido el manual de Linux, disponible mediante el comando *man*. Dicho manual ha tenido dos usos principales.

En primer lugar, ha sido usado como referencia de la sintaxis que esperan las funciones de la *libc*. Las entradas correspondiente a este uso están en la página 3, que corresponde al Manual de Funciones de Librería. Por ejemplo:

```
man 3 snprintf
```

Este comando abre el manual y podemos ver el número y tipo de argumentos que espera la función, la descripción de su funcionamiento, el valor que devuelve, y los códigos de errores asociados. Es una herramienta muy completa.

En segundo lugar, el manual de Linux ha sido utilizado como referencia de la sintaxis y funcionamiento de las llamadas al sistema (o *syscalls*). De esta manera, podemos ser capaces de entender qué hace cada llamada al sistema, el número de argumentos que tiene, su tipo, y las cabeceras necesarias para poder ejecutarla. Las entradas correspondientes a este uso están en la página 2, que corresponde al Manual de Llamadas al Sistema. Por ejemplo:

```
man 2 getuid
```

Este comando abre el manual y muestra la información detallada de la llamada al sistema, indicando el número de argumentos y su tipo, el tipo de variable que devuelve, y los errores que puede retornar. Si ejecutamos `man syscall`, también podemos ver la convención de argumentos de las llamadas al sistema, para ver qué registros contienen argumentos y en qué orden se pasan, además de como se indica el número de *syscall* y donde se devuelve el valor de retorno.

4.5. Otras herramientas

Para el control de versiones de los repositorios, se ha utilizado la herramienta `Git`, almacenando dichos repositorios en GitHub. De igual manera se han utilizado las *releases* de GitHub para el lanzamiento de los binarios en sus diferentes versiones, en los casos de la librería DLL y la GUI para Windows (distribuida en formato `.zip`).

De igual manera, para la realización tanto de esta memoria como de los anexos, se ha empleado la herramienta LaTeX, utilizando Overleaf como editor de texto y compilador de LaTeX. Para la realización de diagramas se han empleado recursos web, como TikzMaker. Estos diagramas han podido ser reutilizados en otras partes del proyecto (como la presentación) gracias al uso de PdfTex e Inkscape, que han sido utilizados para convertir los diagramas primero a PDF, y posteriormente a extensión `.svg` (extensión vectorizada para no perder calidad).

Siguiendo con la documentación, para la realización de manuales como los de las pruebas de concepto y otro tipo de documentos, se ha empleado OpenOffice Writer. Para la presentación destinada a la defensa de este trabajo frente al tribunal, se ha empleado Microsoft PowerPoint, y para la grabación de videos explicativos, se ha empleado OBS Studio, que es una herramienta que permite grabar la pantalla, entre otras funciones.

5. Aspectos relevantes del desarrollo del proyecto

En este capítulo se va a detallar en profundidad el proceso de desarrollo del *software*, las distintas fases de este proceso, y las problemáticas encontradas en cada fase. Esta explicación tiene como motivación ofrecer un razonamiento acerca de la solución planteada a los distintos problemas a los que el alumno ha debido hacer frente durante la tarea de desarrollo.

5.1. Ciclo de vida

El ciclo de vida planteado para el proyecto ha tenido una aproximación en cascada. Es decir, con varias etapas diferenciadas en las que se desarrollan tareas diferentes. Pueden verse las diferentes etapas en la Figura 5.1.

Esta aproximación ha sido seleccionada debido a la naturaleza del proyecto. La particularidad de ser un proyecto a realizar por una sola persona acompaña a la idea de un desarrollo en cascada, ya que otras aproximaciones llegan a ser más beneficiosas en proyectos con más personas. El control de versiones del proyecto se ha realizado mediante un repositorio en GitHub, realizando *commits* de manera preiódica, cada vez que se ha conseguido una solución a un problema o una funcionalidad estable. El repositorio está disponible aquí.

El diagrama seguido en durante el proyecto es el siguiente.

$Análisis/Diseño \longrightarrow Implementación \longrightarrow Pruebas \longrightarrow Documentación$

Figura 5.1: Esquema del ciclo de vida en cascada del proyecto.

5.2. Análisis

Durante la fase de análisis, se estudiaron las principales decisiones estratégicas del proyecto, es decir, aquellas que su importancia es mayúscula, y que una mala elección conllevaría tener que volver a repetir mucho trabajo y perder mucho tiempo.

La primera decisión estratégica fue el lenguaje de programación. Tal como se ha comentado en el capítulo anterior, fue elegido el lenguaje C por su cercanía al ensamblador, por su utilidad para las demostraciones de ciberseguridad en programas vulnerables, pero sobre todo, para una gestión eficiente de la memoria. La gestión de la memoria era de vital importancia, ya que el espacio de direcciones debía ser de 2^{32} bytes (~4GB), y C, a diferencia de otros lenguajes de más alto nivel, ofrece mecanismos para reservar una cantidad tan grande de memoria. Dicha reserva de memoria se realiza sobre memoria virtual, pero la memoria física utilizada es la que pueda utilizar un programa normal, por lo que puede convivir perfectamente con otros procesos sin consumir toda la memoria RAM de un dispositivo. Con una implementación simulada de la MMU, podría reducirse la cantidad de memoria virtual reservada, pero es un asunto complejo ya que las direcciones de memoria específicas utilizadas se conocen en tiempo de ejecución.

En cuanto a las librerías, tal como se ha explicado anteriormente, se eligió la librería Capstone para desensamblar los programas, byte a byte, en secuencias de instrucciones que puedan ser interpretadas por el simulador. Inicialmente se pensó en la posibilidad de desensamblar los programas a mano, pero el tamaño variable de las instrucciones, la cantidad de bytes prefijo y sufijo que pueden llevar (o no) las instrucciones, y sobre todo, la cantidad de líneas que iba a engordar el código, hicieron que se optara por desensamblar utilizando esa librería.

También se contempló la posibilidad de cargar de manera dinámica las librerías necesarias por los programas a ejecutar (sólo con el modo ELF), pero se descartó la idea por la complejidad asociada a la carga de las librerías en el espacio del kernel. En su lugar, se planteó la limitación de que los ficheros ELF deban ser compilados estáticamente.

5.3. Diseño

Durante la fase de diseño, se estudiaron una serie de decisiones que implicarían la estructura de los ficheros del proyecto, y como deberían interactuar entre ellos. También se decidieron las funcionalidades específicas que debía satisfacer el simulador.

Estructura de los ficheros

En cuanto a la estructura, se ideó como un conjunto de ficheros, de manera que existieran distintos módulos con responsabilidad única, es decir, que cada módulo se dedicara a una única tarea. Debido a la naturaleza del lenguaje C, cada módulo consta de un fichero cabecera (extensión .h) y un fichero de implementación (extensión .c).

main / proc es el módulo principal, que contiene la lógica inicial del simulador, y llama a las funciones que deben ser ejecutadas en primera instancia. Se encarga de validar los datos obtenidos por el usuario y de desensamblar las instrucciones del programa. Coordina el resto de módulos.

instr es el módulo que se encarga de la lógica de las instrucciones y de la inicialización de otras estructuras de datos del procesador, como la GDT o la pila.

flags es el módulo que se encarga de la lógica de las banderas, y define funciones para modificarlas y para comprobar su estado.

loader es el módulo que se encarga de cargar el programa en memoria y de realizar otras inicializaciones, cuyos parámetros dependen del fichero ejecutable.

interrupts es el módulo que se encarga de la lógica de las interrupciones.

interface es el módulo que define las funciones para la interfaz gráfica integrada en la terminal. Solo disponible en la rama pensada para Linux.

syscall es el módulo que contiene las implementaciones de las llamadas al sistema. Solo disponible en la rama pensada para Linux.

trad_syscall es el módulo que traduce los números de llamadas al sistema de 32 bits a 64 bits. Solo disponible en la rama pensada para Linux.

typesi386 es el módulo que traduce el tamaño de los tipos y structs en bytes de 32 bits a 64 bits. Necesario para la traducción de las llamadas al sistema. Solo disponible en la rama pensada para Linux.

Estructura de la interfaz gráfica

De cara a la implementación de las interfaces gráficas (ya fuera Linux o Windows), el diseño de la interfaz de usuario era un paso fundamental. Esto se debe a que es la diferencia entre conseguir un programa fácil e intuitivo para el usuario, o una pesadilla para éste.

Por ello, se elaboró un diseño basado en ventanas. Dichas ventanas proveerían distinta información, y corresponderían a los registros, al código, y a la pila. Véase la Figura 5.2.

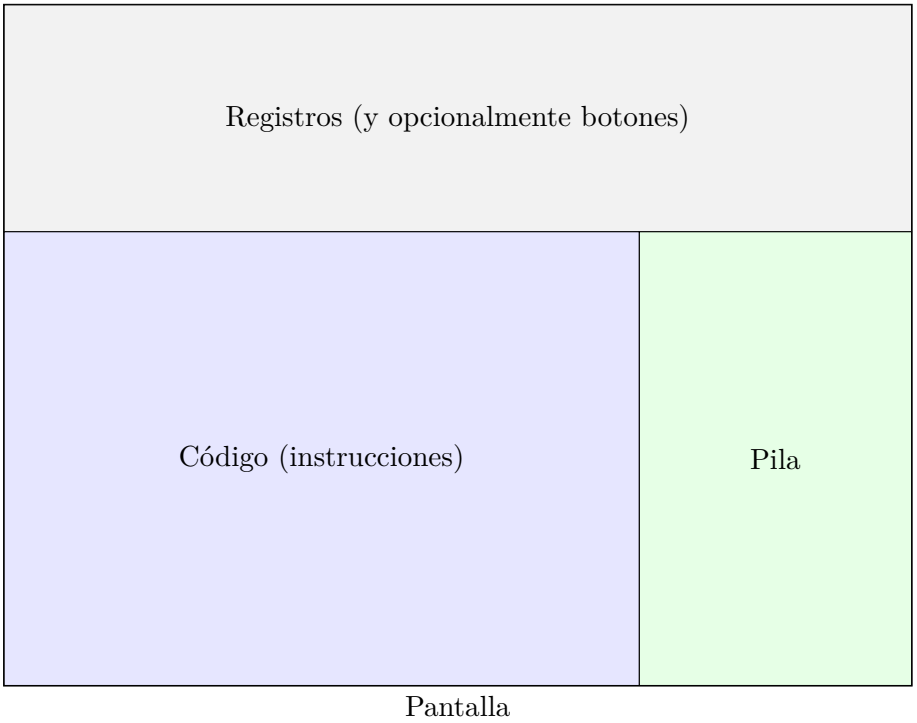

Figura 5.2: Estructura de la interfaz gráfica

5.4. Implementación

Durante la fase de implementación, han salido a relucir varias problemáticas que afectaban al desarrollo, cuya solución no es trivial. A continuación se va a proceder a explicar el quid de algunos de estos problemas y la solución planteada, aunque sean detalladas con más rigor y extensión en los anexos.

Desensamblaje del código

A la hora de cargar el código en memoria, proveniente de un ejecutable en formato ELF, no se carga todo el código a la vez, sino que se cargan varias secciones de los *Section Headers* que tienen la flag *X* (ejecutable). Estas secciones pueden no ser consecutivas en memoria una vez cargadas desde el ejecutable, es decir, que el último byte de una sección no tiene por qué ser contiguo con el primer byte de la siguiente sección. Por ello, no podemos desensamblar todas las instrucciones de un plumazo, ya que entre medias habría bytes no inicializados (indefinidos) que contaminarían el desensamblaje.

Por ello, debe hacerse un desensamblaje por sección, y las instrucciones deben ser almacenadas en el array que corresponde a la sección, de manera que en vez de tener un array con todas las instrucciones, tenemos un array de arrays de instrucciones, donde cada array corresponde a una sección. Esto se ve reflejado en el código como un puntero a puntero, en vez de un puntero.

```
cs_insn **insns = calloc(cc, sizeof(cs_insn *));
```

Donde `cc` es una variable que almacena el número de secciones ejecutables que tiene el ejecutable, y que se obtiene durante la lectura del ejecutable ELF. El acceso o búsqueda de instrucciones también debe hacerse iterando primero entre secciones y por instrucciones en cada sección, lo que obliga a usar dos bucles en vez de uno.

Esta anomalía solo tiene lugar en el Modo ELF. En el modo de ejecución de ficheros *Raw*, sí que es un solo array el que contiene todas las instrucciones, cuya declaración es la siguiente.

```
cs_insn *insn;
```

Ordenación de funciones

Durante la lectura del fichero ELF, obtenemos los nombres de las funciones presentes en el código a partir de la tabla de símbolos, pero estas funciones no vienen necesariamente ordenadas en función de su dirección de memoria.

Como la interfaz accede de manera recurrente a los nombres de las funciones utilizando su dirección de memoria, se ha optado por ordenar los nombres de las funciones (los *strings*) en función de su dirección de memoria, de manera ascendente. De esta manera se reduce la complejidad temporal de buscar la función a la que corresponde una instrucción.

Para la ordenación de estos pares dirección-*string*, se ha optado por la utilización de un *Bubble Sort* sobre la dirección de memoria.

Syscalls en Linux

Tal y como se ha comentado anteriormente, las llamadas al sistema en 32 bits son distintas a las de 64 bits. Por lo tanto, a la hora de querer ejecutar las llamadas al sistema en el simulador, deberemos traducirlas a las llamadas al sistema de 64 bits. Esta traducción depende de varios factores.

Arquitectura las llamadas al sistema cambian, tanto de argumentos como de número identificador en función de si estamos en x86, x86_64 o en arm64. Algunas de hecho pueden ni existir en otras arquitecturas.

Número de syscall los números identificadores de syscall varían de una arquitectura a otra, por lo que deben ser traducidos de x86 a x86_64.

Argumentos cómo se pasan los argumentos, o el tamaño en bytes de estos varían de x86 a x86_64, por lo que deben ser traducidos todos los valores, punteros y structs que se utilicen en las llamadas al sistema.

Instrucciones irrelevantes

Hay un número de instrucciones del i386 que han quedado obsoletas o que no son relevantes en el contexto actual y han quedado sin implementar. Esto se puede deber a que son instrucciones que manipulan estructuras de datos que no existen o no aplican en el simulador, como los registros de *debugging* (i.e. CR0). También se puede deber a que son instrucciones orientadas a establecer una sincronización entre varios procesos, cuando el simulador ha sido creado con una perspectiva monohilo. Otras instrucciones,

simplemente, han caído en desuso y no pueden encontrarse en programas modernos, ya que existen otras menos complejas que las sustituyen.

De igual manera, aproximadamente el 95 % de las instrucciones sí que han sido implementadas y son suficientes para llevar a cabo cualquier tarea.

Traducción de argumentos

En el modo ELF del simulador, se utiliza el array `argv` predefinido por el lenguaje C, así como el parámetro `argc`, que indica el tamaño de `argv`. Este array corresponde al número de argumentos que ha tomado el programa, correspondiendo la posición 0 al nombre del programa y las posiciones siguientes a los argumentos, por lo que `argc` siempre es, como mínimo 1.

Cuando llamamos al simulador, y le especificamos el modo del simulador (i.e -nr) y el ejecutable de 32 bits a ejecutar, estamos utilizando un `argv` de como mínimo 3 posiciones. Sin embargo, a la hora de cargar los argumentos del programa en la pila de la memoria del i386, no podemos cargar todos los argumentos del programa host, sino solo los relevantes al programa de 32 bits que vamos a emular. Por lo que el programa detecta cuál es el último argumento que corresponde a indicaciones del simulador, y carga el resto en memoria. Todos los elementos de `argv`, ya sea en 32 o 64 bits, son strings.

Proceso ./proc (64bits)

Índice	**Valor**
argv[0]	./proc
argv[1]	-nr
argv[2]	-b
argv[3]	0xFFFE0000
argv[4]	-t
argv[5]	0xFFFDC000
argv[6]	prog32
argv[7]	arg1

Programa prog32 (32bits)

Índice	**Valor**
argv[0]	prog32
argv[1]	arg1

Figura 5.3: Traducción de argumentos del programa

5.5. Pruebas

A la hora de realizar las pruebas para comprobar que el comportamiento del *Software* es correcto, se han utilizado las siguientes técnicas. Estas técnicas son prácticamente infalibles ya que, debido a la configuración de los ficheros ejecutables que utiliza el simulador (compilados estáticamente, con *no pie*, y ejecutados con mismo *Stack Range*), la ejecución, tanto en el simulador como en una máquina nativa es completamente determinista y replicable.

Pruebas de flujo con GDB

A la hora de evaluar si la implementación de las instrucciones simuladas era correcta, se ha utilizado el programa GDB. Este programa es un *debugger*, por lo que no ejecuta el programa, solo hace que se ejecute y lo va pausando paso a paso, mostrando los cambios en los registros.

De esta manera, ha sido necesario alinear de manera completa e infalible la pila del programa real, seguido por GDB, con la pila del programa simulado, ejecutado por el simulador. La alineación perfecta de la pila ha sido una tarea ardua que ha requerido replicar de manera fiel como carga el Kernel de Linux la pila al cargar un programa en memoria para su ejecución. Para ello se han debido incluir en la pila estructuras como las variables de entorno, vector auxiliar, argumentos y alineaciones a 16 bits. Que las pilas sean exactamente iguales, es decir, que tengan el mismo *Stack Range*, mismas direcciones de pila para las mismas variables, mismo número y valor de variables de entorno, nos asegura una ejecución simétrica y predecible. Es decir, que los valores de los registros sean exactamente iguales instrucción a instrucción durante todo el programa.

Pudiendo ejecutar el mismo programa de manera nativa y con el simulador, se han programado una serie de *scripts* que han llevado una especie de registro o diario (*Journal*) de los valores de todos los registros para cada instrucción tanto en la ejecución nativa como la simulada. Teniendo ambos diarios, podemos compararlos instrucción a instrucción y ver si los valores de los registros difieren. Esta comparación se ha realizado mediante el desarrollo de un programa personalizado escrito en C. Este programa nos indica en qué línea del diario difieren los valores de algún registro. Sabiendo en qué línea del diario difiere la ejecución, podemos ver el valor del registro EIP en esa línea, poner un *breakpoint* en esa dirección de memoria y ver en qué instrucción se ha obtenido un valor anómalo, y por qué.

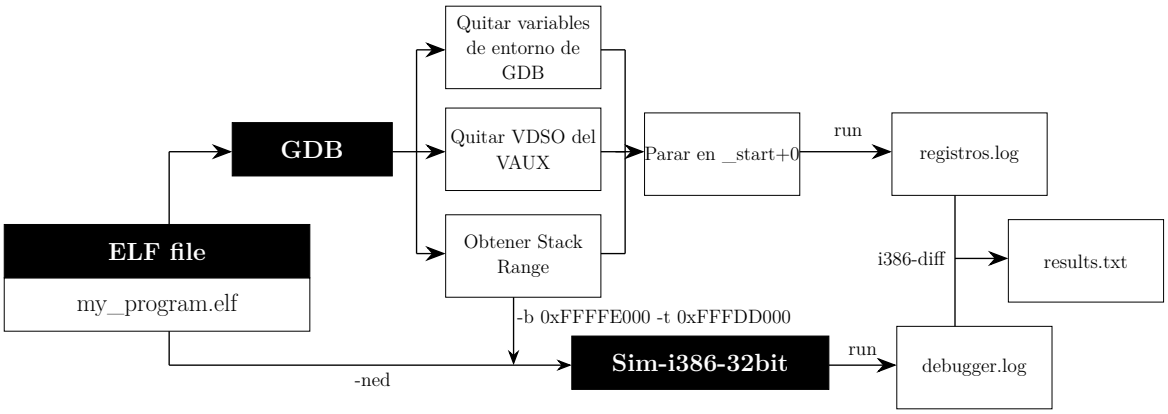

Figura 5.4: Diagrama de flujo de las pruebas realizadas

Por ejemplo, de la ejecución nativa del programa extraeríamos un fichero llamado `registros.log`, mientras que de la ejecución del simulador con la opción `-d` obtendríamos un fichero llamado `debugger.log`. Podemos comparar ambos ficheros mediante la utilización del programa llamado `i386-diff` en la Carpeta debug del repositorio principal.

Este programa produce un resultado con la siguiente estructura.

```
$ ./debug/i386-diff registros.log debugger.log

Line 301: EFLAGS differs: 0x00000216 vs 0x00000287
Line 302: EFLAGS differs: 0x00000286 vs 0x00000287
```

Con esta información podemos ir a la línea justo anterior del diario de la ejecución nativa y ver el valor del registro EIP.

```
$ awk NR==300 registros.log

EIP=0x08048360 EAX=0xffffd41a EBX=0xffffd27c ECX=0x00000000
EDX=0x00000065 ESI=0xffffd168 EDI=0xffffd200 EBP=0x00000000
ESP=0xffffd150 EFLAGS=0x00000216
```

Una vez conocemos el valor del EIP, podemos replicar la ejecución de la simulación parando en esa dirección y ver qué ocurre.

5.6. Entorno Docker para la ejecución del simulador

A juego con el simulador y el resto de herramientas desarrolladas, se ha desarrollado un entorno de ejecución basado en Docker. Este entorno permite emplear el simulador sin tener que realizar instalaciones manuales de las librerías requeridas. La idea principal es que los estudiantes puedan emplear el simulador con ecosistema Linux, sin la necesidad de tener Linux instalado en los ordenadores del laboratorio.

Funciona de manera que el profesor levanta el contenedor del Repositorio del entorno docker para el simulador, incluyendo un fichero CSV con los nombres o alias de los alumnos. El contenedor (que está basado en Ubuntu-22.04) se encarga de descargar una copia del simulador desde el Repositorio original, y compilarla. A continuación, crea tantos usuarios como nombres para ellos haya en el fichero CSV, y crea una copia (enlace simbólico) de los ejecutables necesarios en su directorio personal (`/home/user`). Estos ejecutables son los siguientes:

- Ejecutable del simulador (`proc`).
- *Cross-compiler* para crear programas del i386 (`gcc-i386`).
- Ejecutable para desensamblar programas (`objdump-i386`).
- Ejecutable para leer las cabeceras de programas del i386 (`readelf-i386`).

Por último, activa el demonio `sshd`, que permite conexión por SSH para obtener acceso a la terminal con la orden `ssh`, o subida y descarga de archivos con `sftp`. De esta manera, cada usuario incluido en el CSV tiene su usuario (y contraseña) que les permite acceder de manera remota al entorno del contenedor y ejecutar el simulador. Sin instalaciones, sin cambiar de sistema operativo. Solo empleando una herramienta disponible en cualquier sistema operativo (incluido Windows).

Este proceso está explicado y demostrado en uno de los vídeos explicativos que forman parte de la documentación del proyecto. Podemos encontrar este video como 11-Ejecución-Entorno-Docker.mp4

6. Trabajos relacionados

Actualmente, existen multitud de programas, herramientas y librerías que o bien son capaces de simular programas de manera completa, o que facilitan en gran medida la simulación de estos. A continuación, se van a exponer una serie de programas y herramientas que son relevantes en el estado del arte actual, y en qué se diferencian de este proyecto.

Herramienta	**Propósito**	**Licencia**	**Otros**
GDB	Depuración nativa	GPLv3	Requiere soporte nativo.
QEMU	Emulación funcional	GPLv2	Sin GUI.
Capstone	Desensamblado	BSD	-
Keystone	Compilación	GPLv2	-
Unicorn	Emulación funcional	GPLv2	Sin GUI. Basado en QEMU.
Sim. 8085 UGR	Simulación 16 bits	CC BY-NC-ND 4.0	Orientado a docencia. Empleado en la UBU.

Tabla 6.1: Comparación trabajos relacionados

6.1. GDB (GNU Debugger)

GDB es un programa capaz de depurar otro programa. No es un simulador per se, sino que hace que el sistema operativo ejecute el programa, pero en modo depuración, parándolo y reanudándolo mediante el uso de señales. Permite poder ver el estado de los registros y la memoria paso a paso. Sin embargo, en una máquina donde, por arquitectura o por otros factores, no se pueda ejecutar el programa de manera nativa, no hay depuración

posible. Además, su interfaz es poco intuitiva, y aunque pongamos el modo de ventanas, no podemos ver la pila en tiempo real, sino que tenemos que usar comandos para verla.

Por ello, nuestro simulador constituye una mejora, ya que permite la ejecución (aunque sin llamadas al sistema en arquitecturas no x86_64 sobre Linux) del programa simulado aunque no pudiera ejecutarse de manera nativa, siempre y cuando pueda compilarse para la arquitectura host. Por ejemplo, podríamos ejecutar un programa x86 en arm64, aunque arm64 no pueda ejecutar de manera nativa el programa que vamos a simular.

6.2. QEMU

QEMU es una herramienta de emulación muy potente para todas las arquitecturas, pero que no permite ni ejecución paso a paso ni interfaz gráfica para ver los registros y la memoria. Para poder depurar un programa con QEMU se depende de GDB o de otro depurador.

El simulador del i386 propuesto en este documento no es una herramienta tan potente de simulación, pero ofrece, a diferencia de QEMU, una interfaz intuitiva y amigable, de manera que los usuarios puedan entender lo que está pasando. Por ello, podríamos afirmar que el simulador del i386 ofrece una simulación didáctica, frente a QEMU que ofrece una emulación funcional. Es decir, QEMU se centra en que funcione, mientras que el simulador del i386 se centra en que el usuario entienda cómo funciona.

6.3. Capstone - Keystone - Unicorn

Capstone, Keystone y Unicorn forman un framework de ensamblaje, desensamblaje y ejecución, de manera que las tres librerías colaboran a la hora de conseguir el proceso completo de emulación de un sistema.

- Capstone se encarga del desensamblaje .
- Keystone se encarga del ensamblado.
- Unicorn se encarga de la lógica de ejecución de código emulado.

A pesar de ello, solo se ha utilizado Capstone en el desarrollo del proyecto, ya que el ensamblaje no es estrictamente necesario y la lógica de simulación era más oportuno implementarla de manera propia, permitiendo así un mayor control e integración dentro del simulador y las herramientas que lo utilizan.

6.4. Simulador 8085 de la Universidad de Granada

El simulador del microprocesador 8085 de la Universidad de Granada [2] es un software que permite al usuario realizar una simulación de programas en el microprocesador 8085 desarrollado por Intel. Tiene una interfaz amigable y está orientado a docencia, siendo empleado por la Universidad de Burgos, entre otras. Sin embargo, a diferencia de nuestro simulador del i386, simula un microprocesador de 16 bits.

6.5. Ciberseguridad

En el campo de la ciberseguridad, especialmente en la explotación de programas a bajo nivel, GDB es una de las herramientas más utilizadas para depuración y análisis dinámico de binarios. Sin embargo, su uso puede resultar complejo para principiantes debido a su interfaz de línea de comandos y la necesidad de comprender conceptos de bajo nivel.

Por ello, nuestro simulador ofrece una interfaz más amistosa y fácil de comprender para los usuarios primerizos, de manera que puedan iniciarse en el mundo del *Hacking Ético* con una herramienta que realmente sea una ayuda.

6.6. Conclusión

Existen cantidad de herramientas que consiguen una simulación funcional o que ayudan a ello, pero la esencia del simulador asociado a este proyecto consiste en hacer una herramienta visual, fácil de entender y de modificar para el usuario. En vez de centrarse en que funcione (que también es menester), el objetivo principal del proyecto reside en que el usuario pueda entender cómo funciona, dominando el flujo de los programas paso a paso y entendiendo la explotación que pueda llevarse a cabo en alguno de ellos.
Por ello, tanto para enseñanza de computadores a bajo nivel, como para ciberseguridad al nivel de ensamblador, el simulador es una opción robusta, gratuita y de código abierto que puede ser una herramienta muy útil para docentes y estudiantes.

7. Conclusiones y Líneas de trabajo futuras

Una vez desarrollado el producto de Software, en este caso el simulador, tanto en su variante Linux como en su variante gráfica para Windows, podemos llegar a una serie de conclusiones.

- Hemos desarrollado una herramienta capaz de simular las estructuras de datos y la ejecución de instrucciones del microprocesador Intel 80386. Dicha simulación permite ejecutar programas del i386 en varios formatos y de varios modos, ya sea paso a paso o simulación completa. Nuestra herramienta es un programa fácil de usar, amigable para el usuario, y relata fielmente todo lo que ocurriría dentro de un microprocesador real, desde la información actualizada en la pila hasta las banderas del procesador activas.

- El simulador sirve como laboratorio de pruebas a la hora de realizar la explotación de programas vulnerables en el entorno de Linux, ofreciendo una ejecución replicable y paso a paso.

- El rendimiento es más que satisfactorio, permitiendo ejecutar programas de miles de instrucciones de manera casi instantánea.

- Se han desarrollado herramientas que facilitan la tarea de los docentes a la hora de implementar el simulador en las aulas, permitiendo un entorno idóneo sin la necesidad de instalar el software en los ordenadores del laboratorio.

De manera adicional, como líneas futuras o ampliaciones que puedan realizarse en el marco de este proyecto, podemos destacar una serie de proyectos.

- Implementación de una simulación del MMU, de manera que la memoria reservada para el programa no tenga que ser de 2^{32} bytes, sino que se reserven bloques de bytes dinámicamente, pero requiere un paso extra en la traducción de memoria virtual del programa simulado a memoria virtual del proceso host.
- Desarrollar una función *hash* completa y sin colisiones para los nemotécnicos de las instrucciones, de manera que a partir del nemotécnico y aplicándole esa función *hash*, otengamos el índice de un array donde está el puntero a la función que implementa esa instrucción. Esto permitiría reducir el coste temporal de $\mathcal{O}(n)$ a $\mathcal{O}(1)$ a la hora de determinar qué función de instrucción debe ejecutarse.
- Soporte para llamadas al sistema en la arquitectura arm64, de manera que puedan ejecutarse las llamadas al sistema (INT 0x80) en una máquina host cuya arquitectura sea esta.
- Incluir soporte para programas con enlazado dinámico de librerías.
- Simuladores que implementen versiones posteriores de los microprocesadores de Intel, como el 80486 o el 80686. Estos simuladores podrían utilizar la lógica principal de este proyecto, e incluso gran parte de las implementaciones de las instrucciones.

Bibliografía

[1] R. A. de la Lengua Española. Definición de simulador,ra. Consultado el 19 de abril de 2026. [Online]. Available: https://dle.rae.es/simulador

[2] M. Rodríguez, J. López, and A. Pérez, "Diseño, desarrollo y realización de un simulador del microprocesador 8085," in *Actas de las Jornadas de Enseñanza Universitaria de la Informática (JENUI)*, Granada, España, 2003, departamento de Arquitectura y Tecnología de Computadores.

[3] Wikipedia, "Unidad de gestión de memoria — wikipedia, la enciclopedia libre," 2025, consultado el 7 de junio de 2026. [Online]. Available: https://es.wikipedia.org/w/index.php?title=Unidad_de_gesti%C3%B3n_de_memoria&oldid=164588291

[4] OsDev.org. (2026) Elf (executable and linkable format). Consultado el 4 de junio de 2026. [Online]. Available: https://wiki.osdev.org/ELF

[5] M. Kerrisk. (2025) syscalls(2) — linux system calls. Consultado el 4 de junio de 2026. [Online]. Available: https://man7.org/linux/man-pages/man2/syscall.2.html

[6] MITRE Corporation. (2026) Common weakness enumeration (cwe). Consultado el 4 de junio de 2026. [Online]. Available: https://cwe.mitre.org/

[7] I. T. de Massachussets, "Intel 80386 reference programmer's manual," 2006, consultado el 17 de enero de 2026. [Online]. Available: https://pdos.csail.mit.edu/6.828/2006/readings/i386/toc.htm