Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

De C a ROPLang: un compilador para la automatización de ataques *Return-Oriented Programming*

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor: Alberto Arteaga Moll
Director: Ricardo Julio Rodríguez Fernández

Septiembre de 2026

Agradecimientos

Para empezar, me gustaría agradecer a mi familia cercana, en concreto a mi madre Ana, quien ha hecho posible que pueda realizar tanto este proyecto como este grado, de principio a fin.

También me gustaría agradecer la compañía y amabilidad de mis compañeros de grado, sobre todo la de Juan Antonio Almodóvar Murillo, tan buen compañero de prácticas como de kalimotxo.

Por último, quiero agradecer a Ricardo esta oportunidad única, habiendo sido capaz de captar mi atención con el proyecto más interesante que se me ha ofrecido, y a Eduardo Mena Nieto por su inspiradora combinación de humor y profesionalidad, demostrando que hacer las cosas de otra manera es posible.

Resumen

En las últimas décadas, la seguridad del software se ha convertido en una constante carrera armamentística entre atacantes y defensores. Tradicionalmente, la explotación de vulnerabilidades de corrupción de memoria dependía de la inyección directa de código malicioso para tomar el control del flujo de ejecución de un programa. Como respuesta a estas amenazas, los sistemas modernos implementaron mecanismos de protección y mitigación, tales como la prevención de ejecución de datos, que prohíbe la ejecución de código en regiones de memoria destinadas exclusivamente a datos. Sin embargo, para evadir estas defensas, la comunidad ofensiva evolucionó hacia las técnicas de reutilización de código, las cuales no requieren introducir código nuevo, sino que se aprovechan de código ya presente en el espacio de memoria del proceso.

La programación orientada a retorno es una técnica de ataque de reutilización de código potente y extensamente presente dada la prominencia de lenguajes con gestión de memoria no segura como C. Su desarrollo, sin embargo, es complejo, costoso en tiempo y dependiente del entorno de ejecución. Con el fin de subsanar estas inconveniencias, en este proyecto se realiza un traductor de un lenguaje de alto nivel, preferible por el desarrollador, a un metalenguaje utilizado por una herramienta de búsqueda de código reutilizable que automatice la inyección de un programa en la explotación del ejecutable deseado. Para lograrlo, en este trabajo se describe el compilador desarrollado sobre la herramienta ROP3, discutiendo sus detalles, limitaciones y consecuencias. Este compilador es capaz de traducir operaciones aritméticas, lógicas y de control de flujo escritas en C a `ROPLang`.

Abstract

Over the last decades, software security has become a persistent arms race between attackers and defenders. Traditionally, the exploitation of memory corruption vulnerabilities relied on direct injection of malicious code to hijack the program execution flow. As a response to these threats, modern systems implemented protection and mitigation mechanisms, such as Data Execution Prevention, which forbids code execution in data memory regions. However, to evade these robust defenses, the offensive community evolved towards code reuse techniques, which do not require the introduction of new code, but chain subroutines already present in the process memory space.

Return-oriented programming is a powerful and extensively present code reuse attack technique given the prominence of memory-unsafe languages such as C. Its development, however, is complex, time consuming, and execution environment dependent. Aiming to overcome these drawbacks, this project develops a high level language translator, preferred by the developer, to a metalanguage used by a reusable code search tool in order to automate the injection of a program in the exploitation of the desired executable. In order to achieve this, this dissertation describes the compiler developed for the ROP3 tool, discussing its details, limitations, and implications. This compiler is capable of translating arithmetic, logical and flow-control operations written in C into `ROPLang`.

Índice general

Capítulo 1

Introducción

1.1. Motivación

Entre las técnicas de explotación de vulnerabilidades, una categoría predominante consiste en la reutilización de código, un tipo de ataque que utiliza un flujo de ejecución interceptado para ejecutar fragmentos de código legítimos encadenados de tal manera que realizan un comportamiento deseado por el atacante. Estas técnicas surgieron como respuesta a la contramedida "Write XOR Execute", que define regiones de memoria modificables o ejecutables mutuamente excluyentes. Estas protecciones están extensamente implementadas en los sistemas operativos modernos.

La técnica más prominente en esta categoría es la programación orientada a retorno (comúnmente conocida como ROP por sus siglas en inglés *Return-Oriented Programming*). Esta técnica consiste en unir fragmentos de código que terminan en la instrucción de retorno (denominados *ROP gadgets* o *gadgets* por simplicidad), que devuelven el flujo de ejecución a la dirección almacenada en la pila (manipulada por el atacante y sobre la que se tiene control), lo que permite su encadenamiento (conocido como cadena ROP) para la ejecución de código arbitrario.

Desde su definición en la publicación original de 2007 [1], se han desarrollado numerosos estudios abordando diferentes aspectos, como herramientas de obtención de gadgets, contramedidas, y ofuscación de código [2]. Además, se ha demostrado que esta técnica es Turing completa reiteradamente (es decir, capaz de emular una máquina de Turing) [1]. Los ciberataques, como cualquier otro tipo de aplicación informática, son habitualmente desarrollados en un lenguaje de alto nivel, mientras que los ataques basados en ROP son comparables a un lenguaje ensamblador: instrucciones ejecutables por el sistema, pero de mayor complejidad de análisis del código y, en consecuencia, código más difícil de actualizar y mantener. En este trabajo se pretende cerrar la brecha entre el código de alto nivel y el código utilizable en el ataque.

1.2. Objetivos

El objetivo consiste en desarrollar un traductor de un programa en un lenguaje de alto nivel (concretamente C) a `ROPLang` para su uso en ROP3, mediante el uso de los analizadores léxicos y sintácticos `Flex` y `Bison`, con el fin de soportar el mayor número de características del lenguaje C y, por tanto, ataques de mayor complejidad.

Más específicamente, los objetivos del trabajo son:

1. Implementar el generador de código que produzca el fichero de texto plano en `ROPLang`.
2. Verificar que el fichero en `ROPLang` generado sea sintácticamente aceptado por ROP3 y que la herramienta sea capaz de resolver alguna cadena de gadgets completa.
3. Definir una batería de programas en C que mida la cobertura del subconjunto.

El alcance del trabajo busca dar cobertura a la declaración, asignación y manipulación de variables de los diferentes tipos básicos de C, en concreto; enteros (`char`, `int`), coma flotante y cadenas de caracteres, operaciones aritméticas y lógicas, y control de flujo y bucles.

1.3. Estructura del documento

Este documento está formado por 6 capítulos. El capítulo 2 introduce los conceptos previos de los que parte este proyecto: `ROPLang` y ROP3. El capítulo 3 profundiza en las etapas de desarrollo del compilador y sus limitaciones. El capítulo 4 evalúa la efectividad del flujo de trabajo al aplicarlo en un caso real. El capítulo 5 indaga en el estado del arte y compara cómo se posiciona el compilador frente a otros trabajos relacionados. Finalmente, en el capítulo 6, se reflexiona sobre la efectividad del flujo de trabajo y sus componentes, y se proponen pasos para continuar un trabajo futuro.

Además, se incluyen dos anexos que complementan la documentación del proyecto. El Anexo A recoge los aspectos relacionados con la gestión y planificación del proyecto, mientras que el Anexo B contiene el fichero con las funcionalidades implementadas, junto con su correspondiente traducción a `ROPLang`.

Capítulo 2

Conceptos previos

En este capítulo se explican los conceptos y el trabajo previo del que parte este proyecto, y que se consideran fundamentales para su desarrollo. En concreto, se explican las características del lenguaje `ROPLang` al que se traduce, cómo lo utiliza ROP3 y sus funciones en la cadena de ataque.

2.1. ROPLang

`ROPLang` es un lenguaje virtual que simula operaciones con gadgets [3]. Actúa como una capa de abstracción para ofrecer una interfaz de desarrollo en un conjunto de instrucciones virtuales más reducido, donde cada instrucción tiene un conjunto de gadgets relacionados a los que traducir, aumentando de este modo la capacidad de encontrar gadgets compatibles. Por ejemplo, la instrucción virtual `add(dst, src)` puede ser traducida al gadget `[add dst, src; ret]` o a `[inc dst; ret]`.

Las operaciones nativas del metalenguaje se clasifican en aritméticas (`add`, `sub` y `neg`), de asignación (`mov` y `lc`), de acceso a memoria (`ld` y `st`), lógicas (`xor`, `and`, `or` y `not`), saltos incondicionales (`jmp`) y saltos condicionales (`gcf`, `lsd`, `spa`, `sps`), que incluyen también operaciones de comparación (`eqc` y `ltc`). Este conjunto de operaciones es Turing completo [3] y es similar a un conjunto de instrucciones de un lenguaje ensamblador RISC (*Reduced Instruction Set Computer*).

2.2. ROP3

ROP3 es una herramienta de búsqueda de gadgets en un conjunto de binarios, donde se busca satisfacer una cadena ROP escrita en el metalenguaje `ROPLang` para generar el código (en lenguaje máquina) que provoque que la máquina vulnerable ejecute el ataque [3].

ROP3 es una herramienta de código abierto y con licencia GNU/GPLv3 (disponible en `https://github.com/reverseame/rop3`) desarrollada por el grupo DisCo de la Universidad de Zaragoza. Ofrece soporte nativo a todas las operaciones virtuales definidas en `ROPLang`, además de permitir la definición de otras mediante el uso de YAML.

A pesar de ser relativamente fiel en su implementación de `ROPLang`, existen discrepancias

en las que se profundiza posteriormente. Cabe destacar la ausencia de la instrucción `gcf`, cuya solución se discutirá posteriormente.

2.3. Procesadores de lenguajes `Flex` y `Bison`

Para el desarrollo del trabajo se han seleccionado `Flex` y `Bison`, dos herramientas de código libre ampliamente utilizadas en la construcción de procesadores de lenguajes. Cada una se encarga de una fase diferente del análisis del lenguaje, permitiendo separar el análisis léxico del análisis sintáctico.

2.3.1. `Flex`

`Flex` es una herramienta destinada a la generación de analizadores léxicos. Su función principal es analizar la secuencia de caracteres de entrada y agruparlos en unidades denominadas *tokens*, como identificadores, números, operadores o palabras reservadas [4].

El funcionamiento de `Flex` se basa en la definición de patrones mediante expresiones regulares, a los que se asocian las acciones correspondientes. A partir de estas especificaciones, la herramienta genera automáticamente el analizador léxico.

Se ha escogido `Flex` debido a que permite implementar esta fase de forma sencilla y estructurada, reduciendo la complejidad del desarrollo y facilitando la modificación de las reglas léxicas del lenguaje.

2.3.2. `Bison`

`Bison` es una herramienta utilizada para la generación de analizadores sintácticos a partir de una gramática formal. Su objetivo es comprobar que la secuencia de tokens proporcionada por el analizador léxico cumple las reglas sintácticas definidas para el lenguaje [5].

La gramática se especifica mediante un conjunto de reglas que describen cómo pueden combinarse los distintos elementos del lenguaje. `Bison` utiliza esta información para generar automáticamente el analizador sintáctico y detectar posibles errores durante el análisis.

La elección de `Bison` se debe a que permite expresar la sintaxis del lenguaje de una manera clara y formal, simplificando su implementación y facilitando futuras ampliaciones de la gramática.

Capítulo 3

Compilador desarrollado

En este capítulo se describe el compilador desarrollado, sus características, limitaciones y se profundiza en su contexto dentro del modelo de ataque. Primero, se presenta un modelo de ejecución y desarrollo de un ataque. A continuación, se comentan alternativas consideradas en la fase de análisis y las decisiones de diseño. Por último, se explican detalles de implementación y estructuras auxiliares, y se nombran limitaciones, principalmente del alcance del proyecto.

3.1. Modelo de ataque

Conviene en primer lugar describir el modelo de ataque antes de discutir los detalles del compilador implementado para disponer de una visión global del contexto en el que se desarrolla. Un diagrama del modelo de ataque se expone en la Figura 3.1. Partiendo de la herramienta ROP3, capaz de transformar una cadena ROP en una *payload* utilizable, el papel que desempeña el compilador consiste en transformar un programa escrito en C a uno equivalente expresado en `ROPLang`.

Figura 3.1: Modelo de ataque, resaltando en verde la herramienta desarrollada.

De este modo, un ataque mediante el uso de estas herramientas consiste en el desarrollo del ataque en C, su compilación a `ROPLang` para que ROP3 busque *gadgets* compatibles en los binarios del entorno de ejecución, generando una carga útil (*payload*) a la que, añadida a una sobrecarga que provoque el desbordamiento de pila, efectúe el ataque en la máquina vulnerable.

3.2. Trabajo previo

El desarrollo del compilador se inició durante un periodo de prácticas curriculares realizadas en el Departamento de Informática e Ingeniería de sistemas de la Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza, y se continuó en el presente trabajo. Durante este periodo, se desarrolló un analizador léxico y sintáctico de C como base sobre la que desarrollar las distintas funcionalidades de la herramienta. El modelo del compilador se describe en la Figura 3.2: se trata de una arquitectura clásica orientada a la modularización y abstracción de las diferentes etapas del análisis.

Su tabla de símbolos, descrita en la Figura 3.3, sigue también una estructuración clásica en ámbitos, en la que los identificadores de los ámbitos padres son visibles desde el ámbito actual, mientras que los definidos en ámbitos hijos no son accesibles desde sus ámbitos padres.

Figura 3.2: Arquitectura del compilador.

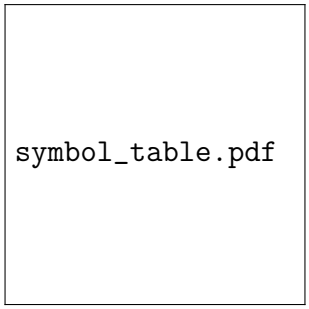

Figura 3.3: Estructura de la tabla de símbolos y visibilidad de ámbito.

Los símbolos almacenados están formados por su nombre y su tipo, soportando los tipos primitivos de C. Esta decisión, aunque habitual en el diseño de compiladores, experimenta una limitación por el funcionamiento de ROP3: dos variables con el mismo nombre son interpretadas como la misma variable, independientemente del ámbito, lo que puede dificultar la búsqueda de *gadgets* al aumentar el número de *gadgets* necesarios para una variable al considerar el conjunto como unidad, a pesar de no ser una relación necesaria, incluso habitualmente incorrecta semánticamente. Para reducir esta limitación, se ha introducido un identificador a las variables auxiliares controladas por el analizador, pero no a las declaradas por el usuario, dado que las colisiones pueden generar confusiones en la tabla de símbolos (por ejemplo, en el caso de `x11`, no es posible distinguir entre `x1` con el identificador `1` y `x` con el identificador `11`).

3.3. Análisis y diseño

El objetivo de este trabajo consiste en la traducción de C a `ROPLang`, por lo que las reglas léxicas y sintácticas de C deben respetarse durante el análisis de entrada y, por esa razón, se ha optado por un diseño de analizador clásico, donde las debilidades y fortalezas son conocidas. En concreto, resultan de especial interés la imposición de reglas sintácticas y la naturaleza recursiva (que permite gestionar niveles arbitrarios de anidamiento), ambas soportadas de manera nativa, en contraste con un enfoque de programación puramente imperativa, en el que estas características deben implementarse de forma explícita.

Al tratarse de un lenguaje ampliamente establecido y con una especificación extensamente documentada como C, se ha considerado oportuna la reutilización de los mecanismos necesarios para su análisis en recursos existentes y de libre uso. En consecuencia, las decisiones de diseño e implementación se han basado en referencias consolidadas, empleando la sintaxis de GNU [6] para tipos y variables, y Microsoft Learn [7] para expresiones.

Dentro del contexto de ROP, al formar parte de la *payload*, las constantes del programa suponen las siguientes implicaciones para el compilador:

- Se introduce una dependencia con el número de bits de la arquitectura; aunque las instrucciones de `ROPLang` no sean dependientes del número de bits, el tamaño de las constantes depende de este número al formar parte de la pila. Este aspecto es configurable mediante una constante del compilador.
- Las cadenas ROP son, en última instancia, cadenas de caracteres como entradas a un programa. En algunos ataques, si una constante contiene caracteres inválidos

(`0x00`, `0x0d`, `0x0a`, `0xff`), la cadena se corta abruptamente y el ataque fracasa. Esta limitación puede no tener solución absoluta, por lo que tampoco es automatizable, especialmente considerando la variabilidad en los entornos de ejecución, en concreto los gadgets disponibles.

Por último, todas las traducciones de instrucciones de alto nivel se han realizado utilizando exclusivamente las operaciones básicas definidas en `ROPLang` [3]. Gracias a esta restricción, se mantiene la capacidad computacional demostrada en [3] al no suponer coste añadido en la obtención de nuevos gadgets, a cambio de un aumento en la complejidad del código generado y, en teoría, unas cadenas ROP con mayor número de instrucciones.

3.4. Implementación

Un aspecto importante a destacar durante esta fase es un especial enfoque en la modularización y reutilización de funciones, además del desarrollo incremental. En el caso de herramientas como `Bison`, estos aspectos son de especial importancia dado que el código es altamente reutilizable; por ejemplo, en las expresiones sintácticas se definen una multitud de operaciones binarias que siguen una lógica similar: el nombre de la operación seguido de los operandos. Así mismo, el desarrollo incremental ha demostrado tener un papel decisivo en la alta dependencia entre las diferentes operaciones (por ejemplo, una multiplicación supone un bucle).

3.4.1. Aritmética

La operativa aritmética está recogida en una función responsable de las siguientes tareas:

- Generar el código adecuado para el tipo de cálculo de la operación y operandos correspondientes: distinguir operaciones a ejecutar de manera simbólica en tiempo de ejecución, utilizar registros auxiliares o calcular el valor resultante en tiempo de compilación.
- En el caso de operaciones simbólicas, los valores son almacenados en registros intermedios con el fin de respetar los valores de los símbolos, dado que las instrucciones de `ROPLang` actualizan el operando izquierdo (similar a la sintaxis de ensamblador Intel).
- Actualizar el tipo de la expresión resultante, ya sea mediante promoción (el tipo de mayor rango toma preferencia) o por efectos secundarios del tipo de operación (operaciones lógicas).
- Distinguir el tipo de operación y utilizar los operadores adecuados. Si la operación es entre constantes, el resultado se calcula con el operador que las define, pero si la operación es simbólica, existen tres distinciones en función del tipo de operación:
    1. La operación es lógica y soportada por la instrucción `gcf`, en cuyo caso se utiliza la instrucción de manera directa (con sus adecuados parámetros). Esta instrucción es necesaria en la implementación de instrucciones de control de flujo. Los operadores no soportados son implementados mediante reglas de equivalencia lógica (por ejemplo, $a \geq b$ es equivalente a $\neg(a < b)$), pero no como parte de esta función.

2. La operación es multiplicativa (multiplicación, división, módulo, desplazamiento lógico). Al no disponer de las instrucciones en `ROPLang`, su implementación requiere un esfuerzo adicional, basado en estructuras de control de flujo. Por esta razón, su desarrollo se ha realizado en las etapas finales del proyecto, una vez que se implementaron todas las operaciones necesarias.
3. En caso contrario, la operación se asume como parte del conjunto de instrucciones de `ROPLang` y se imprime tal cual, junto con sus operandos.

No todas las operaciones se rigen por esta función, pero sí lo hace la mayoría. Un contraejemplo trivial son las operaciones unarias, que disponen de un único operando. Otro ejemplo son los operadores lógicos cortocircuitados, que utilizan una lógica ligeramente diferente.

3.4.2. Control de flujo

Disponiendo de operadores lógicos, y haciendo uso de las operaciones `ROPLang` necesarias, se replican los saltos a direcciones de memoria, condicionales e incondicionales, tal y como se describe en [3]. Sin embargo, aparece una limitación importante que supone una barrera en el desarrollo: las operaciones condicionales requieren una constante δ, que es un inmediato que requiere conocer el número de instrucciones a saltar, con lo que una generación de código en el instante, sin almacenar las instrucciones generadas en ningún momento, no es viable. Tampoco es posible distinguir en qué momento comienza el bloque de instrucciones, ni se puede hacer una estimación a priori por la naturaleza recursiva de la sintaxis. Por todo ello, es necesario replantear el modo en el que se genera el código para dar soporte a esta funcionalidad.

Fundamentalmente, se ha de atrasar la generación de código para calcular δ, y también se han de agrupar los distintos bloques de código según su estructura semántica para delimitar el ámbito de cada δ. Se proponen las siguientes alternativas:

- Utilizar una estructura auxiliar global, similar a la tabla de símbolos, en la que almacenar la generación de instrucciones atrasada para su utilización en el cálculo de δ.
- Modificar los valores que toman todos los elementos de la sintaxis para almacenar las instrucciones generadas hasta el momento y vaciar el búfer en momentos oportunos.

Ante estas opciones, se plantea un compromiso entre una estructuración más compatible con la arquitectura actual, en concreto evitar complicar aquellos elementos con valores ya definidos (como las expresiones), evitar sobrecarga en la propagación de la estructura, y estructurar los criterios de adecuado vaciado del búfer, frente a una estructuración directa del código que evita efectos secundarios y respeta el principio de abstracción. Si bien estas ventajas presentan ciertos beneficios, se ha considerado que no compensan las ventajas que proporciona la implementación de una estructura global. Esta solución permite mantener el estado actual del compilador y preservar una semántica bien definida para los elementos sintácticos. Aunque ello implica una mayor complejidad en las funciones que dependen de dicha estructura, se ha considerado que este coste resulta asumible frente a las ventajas obtenidas. Por tanto, se ha optado por esta alternativa.

La naturaleza de este búfer es la de una pila: en el anidamiento de condicionales, todas las instrucciones relacionadas con el condicional más reciente forman parte del condicional

inmediatamente anterior a éste.

Para introducir elementos en esta pila, se hace uso de una función que recibe un número de parámetros variable (basada en el modelo de la función de biblioteca estándar de C `printf`), siguiendo un formato para su posterior procesamiento, en el caso de que este fuera necesario.

El formato de la síntesis de instrucciones consiste en una separación de salto de línea por cada instrucción de `ROPLang`, y un salto de línea adicional si se trata del fin de una instrucción de alto nivel. De este modo, contando el número de saltos de línea no consecutivos dentro de un bloque de código, multiplicado por el número de bytes de la arquitectura, se obtiene el valor de δ necesario para ejecutar los saltos de control de flujo.

A modo de ejemplo, en las Figuras 3.4 y 3.5 se muestran detalladamente los pasos tomados por el compilador para generar una instrucción `if`. Cuando el analizador detecta el paréntesis de apertura de la instrucción `if`, genera un nuevo nivel donde almacenar el código de la condición de la instrucción. Una vez que ha terminado (encuentra el paréntesis de cierre de la instrucción `if`), apila otro nivel donde se retienen las instrucciones correspondientes a una evaluación cierta de la condición. Tras procesar el bloque entero, el analizador desapila este bloque de código, calcula δ, introduce las instrucciones `lsd`, `spa` y el código generado, y lo añade al bloque anterior, donde se encuentra la instrucción `if` traducida y, si es el último bloque de la pila, genera el código de salida.

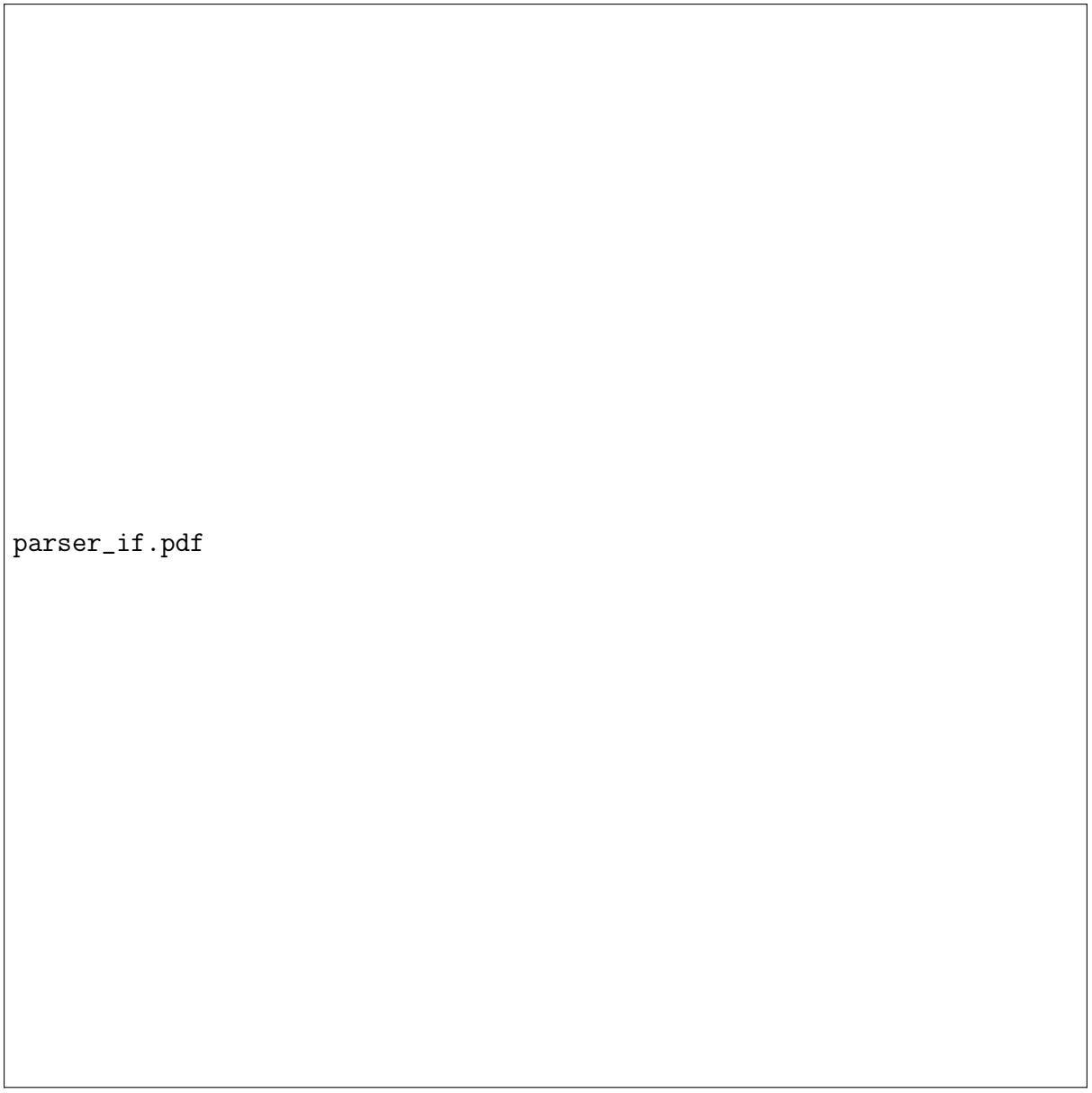

Figura 3.4: Traducción de la instrucción `if`, paso 1: procesamiento sintáctico. En azul, la posición del analizador.

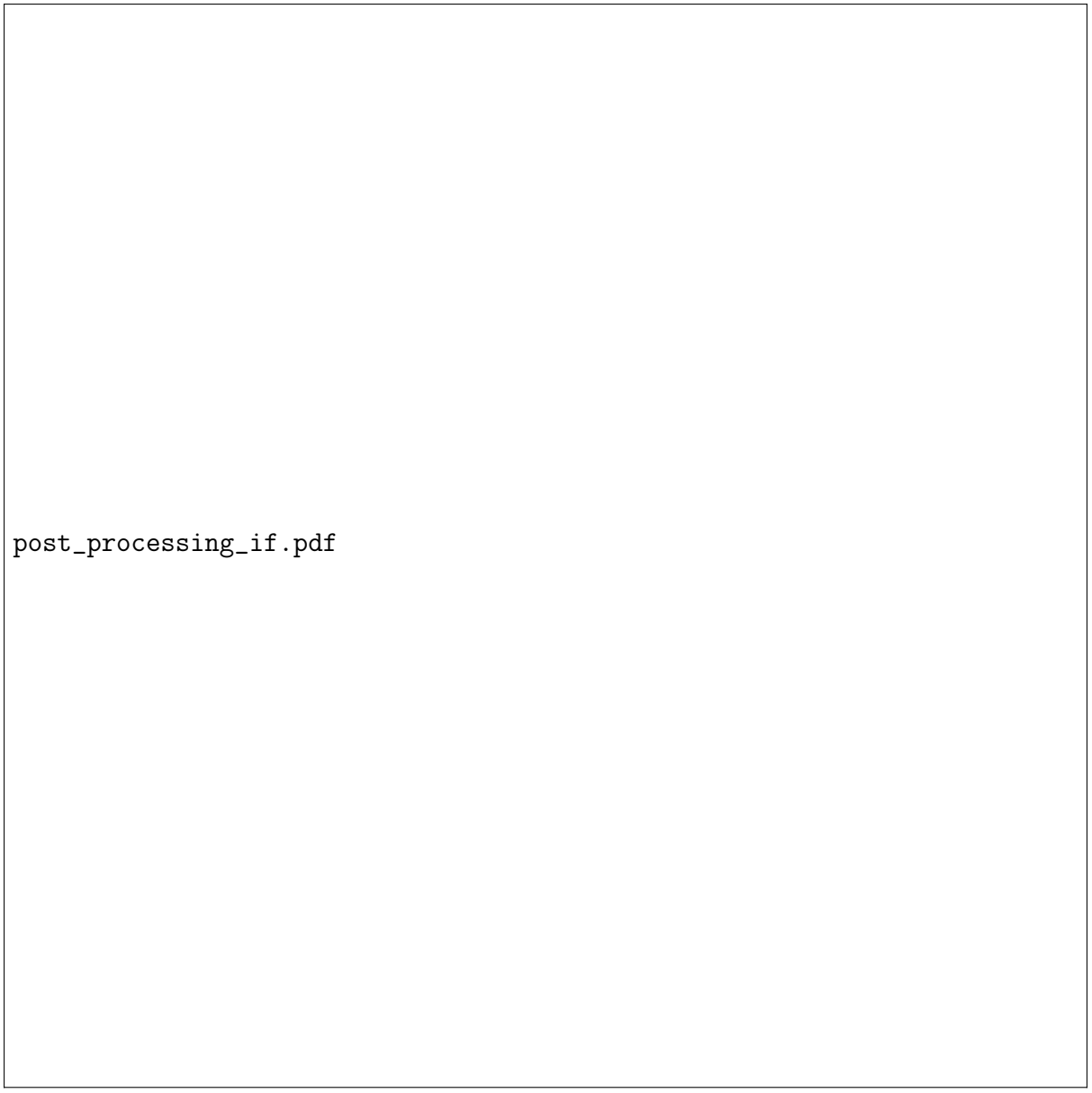

Figura 3.5: Traducción de la instrucción `if`, paso 2: generación de código.

Un efecto secundario de esta estructura en pila es su agrupamiento de distintos bloques de código, que permite reordenarlos según se desee. Esta modularización mejora la escalabilidad de código y la reutilización del mismo; un ejemplo donde se hace uso de esta técnica es la construcción del bucle `for` (Figura 3.6), donde se adapta el procesamiento sintáctico para la reutilización de la generación de código del bucle `while`. Un bucle `for` está formado por 4 elementos: inicialización, condición del bucle, paso y cuerpo del bucle. Dos de ellos, la condición y el cuerpo del bucle, forman parte de la estructura de un bucle `while`, la inicialización se ejecuta una vez antes del bucle, por lo que ese código no forma parte del bucle, y el paso se ejecuta al final del cuerpo del bucle. Para preparar la pila a esta estructura, se desapilan todos los bloques, se genera el código de inicialización y se añade el paso al final del cuerpo del bucle. y se vuelven a apilar ambos bloques en el bucle.

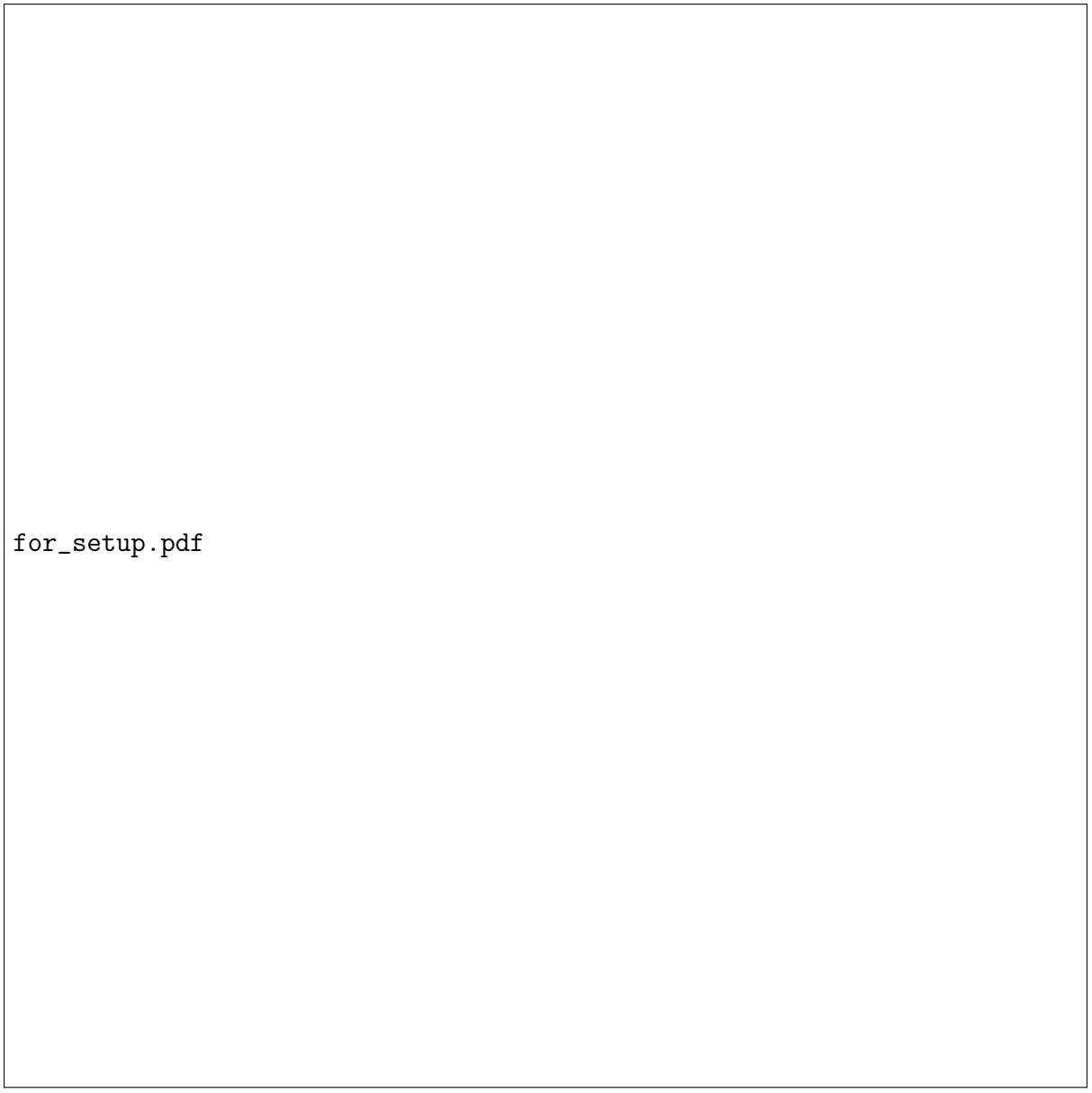

Figura 3.6: Procesamiento sintáctico de la instrucción `for`: equivalencia con `while`, pasos sintácticos y adaptación del estado final de la pila.

3.5. Limitaciones

Dado que el foco del desarrollo ha sido el soporte de expresiones y el control de flujo, no se ha invertido esfuerzo en el control estricto de tipos y los tipos compuestos (vectores y tipos personalizados `struct`), aunque sí existe soporte para declarar punteros (actuando como enteros) a cadenas de caracteres mediante literales (con comillas dobles). Tampoco existe soporte para punteros de otro tipo y funciones.

Como se ha mencionado anteriormente, se ha planteado como problema abierto el saneamiento de caracteres inválidos en la cadena ROP. La colisión de nombres iguales sí ha sido subsanada, aunque la solución utiliza identificadores y podría mejorarse con técnicas más avanzadas como optimización de código.

Capítulo 4

Evaluación

El presente capítulo tiene como objetivo evaluar en qué medida esta implementación satisface los objetivos del trabajo, siguiendo el proceso de la traducción de C a `ROPLang`: si el código `ROPLang` generado es sintácticamente válido y si ROP3 es capaz de resolver a partir de él una cadena de gadgets completa sobre un binario objetivo. No es de interés experimentar con la cadena sobre el binario, ya que dicho comportamiento es exclusivo de ROP3 y supondría un esfuerzo adicional para unos experimentos que no son de interés.

Para ello, en primer lugar se describe el entorno de experimentación empleado (versiones de software, binario(s) objetivo y máquina de pruebas), de forma que los resultados sean reproducibles. Después, se presentan los casos de estudio diseñados para cubrir el subconjunto de C descrito. Por último, se discuten los resultados obtenidos y las limitaciones detectadas.

4.1. Entorno de experimentación

El desarrollo y las pruebas del traductor se han realizado utilizando ROP3 versión 1.0.0, con una extensión de la implementación de `ROPLang` para soportar instrucciones del tipo `gcf`. Este entorno permite ejecutar y comprobar el código `ROPLang` generado por el traductor.

La máquina de pruebas utiliza el sistema operativo Debian 13 y presenta una arquitectura x86-64.

4.2. Casos de estudio

Con el objetivo de evaluar el funcionamiento del compilador y de la herramienta ROP3, se ha diseñado un conjunto de pruebas que permite verificar de forma progresiva las funcionalidades implementadas. En primer lugar, se ha elaborado un fichero que contiene el conjunto de instrucciones y operaciones soportadas por el compilador. Esto permite comprobar individualmente la correcta implementación de las distintas funcionalidades y detectar posibles errores durante el proceso de desarrollo. Dado el tamaño del fichero, se encuentra adjunto en Código B.1.

Posteriormente, se ha empleado un programa encargado de calcular el factorial de un

número como caso de prueba más representativo de un caso real. De esta forma, se puede comprobar el comportamiento del compilador sobre un programa con la lógica de una utilidad existente (Código 4.1).

Finalmente, se ha desarrollado un programa sencillo que realiza una operación de suma con el propósito de disponer de un ejemplo capaz de utilizarlo como entrada para la herramienta ROP3. Este último caso permite analizar el comportamiento de la herramienta sobre la salida generada y comprobar su capacidad para localizar y obtener los gadgets disponibles, sirviendo como prueba de integración entre el código generado por el compilador y el proceso de análisis realizado por ROP3 (Código 4.2).

```c
//#include <stdio.h>

int main() {
  int N, fact, i;
  N = 5;
  fact = 1;
  for (i = 1; i <= N; i++) {
    fact = fact * i;
  }
  //printf("Factorial of %d is %d", N, fact);
  return 0;
}
```

Código 4.1: Fichero `factorial.c`.

```c
int main() {
  int a, b, c;
  a = 5;
  b = 10;
  c = a + b;
  //printf("Sum of %d and %d is %d", a, b, c);
  return 0;
}
```

Código 4.2: Fichero `suma.c`.

4.3. Discusión de resultados

Los resultados obtenidos muestran que el compilador desarrollado es capaz de traducir programas sencillos escritos en C a instrucciones de `ROPLang`, manteniendo la lógica del programa original. De igual manera, dado el tamaño del fichero, la traducción del fichero con las funcionalidades implementadas se encuentra en el Código B.2.

En el caso de `suma.c`, la traducción es especialmente sencilla. Como se muestra en el Código 4.4, el programa requiere únicamente la inicialización de tres variables y una operación de suma. El código generado utiliza los registros asociados a las variables `a`, `b` y `c`, además de un registro temporal, `regResult0`, utilizado para realizar la operación. En total, se generan 10 instrucciones de `ROPLang`, incluyendo las instrucciones de carga, movimiento y suma. Este resultado muestra que las operaciones aritméticas básicas pueden traducirse de forma relativamente directa, requiriendo pocos registros temporales y, por tanto, una cadena ROP de complejidad reducida.

Por otro lado, `factorial.c` presenta una traducción considerablemente más compleja, expuesta en el Código 4.3. En este caso se utilizan los registros asociados a las variables `N`, `fact` e `i`, junto con varios registros temporales (`regResult0`, `regResult1`, `regResult2`, `regResult4`, `regResult5`, `regResult7` y `regResult8`) y registros auxiliares como `regRight`, `regiMul` y `regaux3`/`regaux6`. El listado generado contiene 38 líneas de instrucciones, aunque parte de ellas corresponden a los valores constantes asociados a las instrucciones `lc`. Este incremento respecto al ejemplo de la suma se debe principalmente a la necesidad de implementar el control de flujo del bucle y la operación de multiplicación mediante varias instrucciones.

Estos resultados ponen de manifiesto una de las principales ventajas de la aproximación propuesta: el programador puede expresar la lógica del programa utilizando construcciones de un lenguaje de alto nivel, mientras que la gestión de los registros y la composición de las operaciones necesarias se delega en el compilador.

```
1
2  lc(regN)
3  0x0000000000000005
4
5  lc(regfact)
6  0x0000000000000001
7
8  lc(regi)
9  0x0000000000000001
10 mov(regResult0, regN)
11 gcf-ltc(regResult0, regi)
12 mov(regResult1, regResult0)
13 lc(regRight)
14 0x0000000000000001
15 xor(regResult1, regRight)
16 mov(regResult8, regResult1)
17 lsd(regResult8, 152)
18 spa(regResult8)
19 mov(regResult4, regfact)
20 lc(regiMul)
21 0x0000000000000000
22 mov(regResult5, regiMul)
23 gcf-ltc(regResult5, regi)
24 mov(regResult7, regResult5)
25 lsd(regResult7, 40)
26 spa(regResult7)
27 add(regResult4, regfact)
28 lc(regaux6)
29 0x0000000000000001
30 add(regiMul, regaux6)
31 spa(, -72)
32 mov(regfact, regResult4)
33
34 mov(regResult2, regi)
35 lc(regaux3)
36 0x0000000000000001
37 add(regi, regaux3)
38 spa(, -216)
```

Código 4.3: Traducción a `ROPLang` de `factorial.c`.

```
1
2 lc(rega)
3 0x0000000000000005
4
5 lc(regb)
6 0x000000000000000a
7
8 mov(regResult0, rega)
9 add(regResult0, regb)
10 mov(regc, regResult0)
```

Código 4.4: Traducción a `ROPLang` de `suma.c`.

4.4. Limitaciones detectadas

Los experimentos sufren una limitación substancial por el limitado número de registros disponibles en la práctica: 8 en x86, mientras que una implementación fiel de C requiere más registros, y sería necesaria una optimización a posteriori para su explotación mediante ROP3. Esta limitación es el principal cuello de botella del traductor: un programa sencillo como un factorial requiere, sin optimización, más de 10 registros. A modo de contraste, el fichero de suma sí es procesable por ROP3 dado que el número de registros utilizados es 4.

Otra limitación detectada durante esta etapa es una discrepancia fundamental entre las dos herramientas: ROP3 es una herramienta de búsqueda de *gadgets* y únicamente espera *gadgets* como entrada. Sin embargo, para la generación de código de una cadena ROP, tanto las constantes como su posición son necesarias. Dado que el uso conjunto de las dos herramientas es deseado, es de interés mejorar la compatibilidad entre ellas. Por ejemplo, ROP3 podría permitir valores de constantes en su entrada de cadenas ROP, aunque no haga uso de ellos durante su ejecución. Actualmente, es posible decidir desde el traductor si se desean imprimir las constantes mediante una variable.

A pesar de tratarse de una implementación de `ROPLang`, en la práctica, existen diferencias y limitaciones en ROP3. En concreto, la instrucción `gcf`, a pesar de ser una instrucción de `ROPLang` definida en [3], no está implementada en ROP3, por lo que se deben añadir los archivos YAML correspondientes a las instrucciones `gcf` que utilicen `eqc` y `ltc`.

Además, en ROP3, los nombres de los registros abstractos deben ser prefijados por "reg" y no admiten guiones bajos, por lo que ha sido necesario modificar la tabla de símbolos para actuar como interfaz a los nombres de las variables, almacenando internamente los mismos pero imprimiendo nombres de registros.

El control de flujo en ROP3, aunque también en la programación orientada a retorno en general, resulta conflictivo dada la ausencia de *gadgets* con modificaciones de pila relativas de valores arbitrarios. Sí es habitual la existencia de modificaciones de pila relativas (aunque requieren encadenamientos no triviales de las mismas), suelen estar limitadas a múltiplos de 8 y son dependientes del entorno de ejecución, información que no es accesible en tiempo de compilación. Por lo tanto, aunque en teoría no resulta un problema, en la práctica se trata de un problema abierto, y es el otro considerable cuello de botella en la automatización, junto con el número de registros disponibles. De hecho,

en versiones más recientes de ROP3, el control de flujo se realiza mediante posiciones de memoria conocidas, información que tampoco es accesible en tiempo de compilación.

Capítulo 5

Trabajo relacionado

En este capítulo se discute el estado del arte y la posición del trabajo desarrollado en relación a los mismos, cómo se diferencia y qué ventajas aporta en su comparación.

Varios trabajos han abordado la generación automática de cadenas ROP a partir de una descripción de más alto nivel que la del propio ensamblador [8]. Schwartz, Avgerinos y Brumley [9] presentaron Q, un sistema que, mediante técnicas de verificación semántica de programas, clasifica los gadgets disponibles en un binario y compila hacia ellos un lenguaje propio, logrando generar *payloads* funcionales para una parte significativa de los binarios de un sistema Linux típico incluso cuando una porción reducida del código permanece sin aleatorizar por ASLR [10]. En una línea similar, ROPC [11] compila un lenguaje de alto nivel propio, ROPL, hacia una cadena de gadgets, clasificando previamente los candidatos y verificando su semántica mediante un solucionador SAT (problema de satisfacibilidad booleana), que permite comprobar automáticamente el cumplimiento de las restricciones definidas; su enfoque de compilación introduce el concepto de pseudo-instrucciones que se desenrollan en listas de *gadgets* evitando conflictos entre registros. Trabajos más recientes, como MAJORCA [12], generalizan el problema a múltiples arquitecturas (x86 y MIPS) y a cadenas mixtas ROP/JOP [13], incorporando además el tratamiento de símbolos restringidos en direcciones y datos, una limitación práctica que los primeros compiladores no cubrían con detalle. Existen igualmente aproximaciones basadas en ejecución simbólica y motores como angr [14] para sintetizar cadenas ROP de forma dirigida por restricciones, así como herramientas orientadas a la explotación práctica, como ROP ROCKET [15], centradas en ampliar la superficie de ataque más que en ofrecer un lenguaje de programación completo.

Un rasgo común a la mayoría de estos compiladores es que su lenguaje fuente es un lenguaje propio, diseñado *ad hoc* y de bajo nivel, próximo en expresividad al propio conjunto de *gadgets*. Esto simplifica la traducción, pero traslada al usuario la carga de aprender una sintaxis específica de la herramienta y de expresar su lógica de explotación en términos poco familiares para un programador habituado a lenguajes de propósito general.

A diferencia de los compiladores descritos, que traducen hacia su propio lenguaje intermedio de bajo nivel de manera acoplada a la búsqueda de gadgets, este TFG aborda la compilación desde un lenguaje de propósito general y ampliamente conocido –un subconjunto de C– hacia `ROPLang`, un lenguaje intermedio ya definido, formalizado y con una

herramienta de materialización (ROP3) independiente y publicada [3]. Este planteamiento presenta dos ventajas respecto al estado del arte revisado. En primer lugar, reduce la barrera de entrada para describir la lógica de una cadena de explotación, al permitir expresarla en un lenguaje con estructuras de control, tipos y expresiones familiares para cualquier programador, en lugar de en un ensamblador virtual o en un lenguaje propio de la herramienta. En segundo lugar, al generar `ROPLang` en vez de *gadgets* concretos, el compilador desarrollado se mantiene desacoplado del binario vulnerable final: el mismo programa fuente en C puede reutilizarse frente a distintos binarios sin más que volver a ejecutar ROP3 sobre el `ROPLang` generado, en lugar de repetir la fase de compilación completa.

Hasta donde alcanza esta revisión, no existe en la literatura un compilador que tome como entrada un lenguaje de propósito general como C y produzca como salida programas en `ROPLang`, lo que constituye el hueco que este trabajo cubre.

Capítulo 6

Conclusiones y trabajo futuro

Las técnicas de programación orientada a retorno constituyen una estrategia de explotación que permite construir cadenas de ejecución reutilizando fragmentos de código ya presentes en un programa, conocidos como *gadgets*. La construcción de estas cadenas puede resultar compleja y laboriosa, especialmente cuando aumenta su longitud o complejidad. En este contexto, `ROPLang` plantea la posibilidad de expresar las operaciones deseadas mediante un lenguaje de mayor nivel y simplificar la generación de cargas útiles, reduciendo así el esfuerzo necesario para su desarrollo.

En este trabajo, se ha estudiado la utilización de `ROPLang` para la construcción automática de cadenas ROP a partir de un lenguaje de alto nivel, con el objetivo de agilizar la fase de desarrollo y facilitar el descubrimiento de estrategias de defensa efectivas ante potenciales ataques ROP.

A pesar de las limitaciones del compilador desarrollado, es factible su utilización para reducir el esfuerzo en la traducción de las instrucciones más fundamentales de un programa en C a `ROPLang`. El uso de `ROPLang` está, sin embargo, sujeto a las limitaciones de ROP3 y es necesario mejorar la integración entre ambas herramientas.

En relación con el trabajo futuro, los esfuerzos principales han de centrarse en cubrir las limitaciones actuales del compilador finalizando su desarrollo, especialmente el soporte a funciones y tipos complejos. Finalmente, es necesario la implementación de un filtrado posterior de caracteres inválidos para obtener cadenas alternativas que sean válidas.

Bibliografía

[1] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 552–561, New York, NY, USA, 2007. Association for Computing Machinery.

[2] Giulio De Pasquale, Fukutomo Nakanishi, Daniele Ferla, and Lorenzo Cavallaro. ROPfuscator: Robust Obfuscation with ROP. In *2023 IEEE Security and Privacy Workshops (SPW)*, pages 1–10, 2023.

[3] Daniel Uroz and Ricardo J. Rodríguez. Evaluation of the Executional Power in Windows using Return Oriented Programming. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 361–372, 2021.

[4] Vern Paxson. Flex - The Fast Lexical Analyzer Generator. `https://github.com/westes/flex`. Accedido: 1 de septiembre de 2026.

[5] GNU Project. GNU Bison Manual. `https://www.gnu.org/software/bison/`. Accedido: 1 de septiembre de 2026.

[6] GNU Project. GNU C Reference Manual. `https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html`. Accedido: 1 de septiembre de 2026.

[7] Microsoft. Microsoft Learn's C Language Reference. `https://learn.microsoft.com/en-us/cpp/c-language/c-language-reference`. Accedido: 1 de septiembre de 2026.

[8] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, 15(1):1–34, 2012.

[9] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security '11)*. USENIX Association, 2011.

[10] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 298–307. ACM, 2004.

[11] pakt. ROPC: A Turing Complete ROP Compiler. `https://github.com/pakt/ropc`, 2013. Accedido: 1 de septiembre de 2026.

[12] Alexey Nurmukhametov, Alexey Vishnyakov, Vlada Logunova, and Shamil Kurmangaleev. MAJORCA: Multi-Architecture JOP and ROP Chain Assembler. In *2021 Ivannikov Ispras Open Conference (ISPRAS)*, pages 37–46. IEEE, 2021.

[13] Thomas Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

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

[15] Bw3ll. ROP ROCKET: An Advanced Code-Reuse Attack Framework. `https://github.com/Bw3ll/ROP_ROCKET`, 2023. Accedido: 1 de septiembre de 2026.

Anexo A

Gestión del proyecto

En el presente anexo se recoge información adicional sobre el desarrollo del proyecto, con el objetivo de aportar una visión más detallada de su evolución y de la dedicación empleada en las diferentes actividades.

Con el fin de facilitar la comprensión de la planificación temporal del proyecto, en la Figura A.1 se presenta su diagrama de Gantt, en el que se muestran las distintas tareas realizadas, su duración y el orden en que se han llevado a cabo. El proyecto se divide en cuatro etapas fundamentales, donde ha existido solapamiento en aquellas tareas de mayor importancia; en concreto, el desarrollo del compilador y la redacción de la memoria. Además, en la Tabla A.1 se detalla la cantidad de esfuerzos dedicados a cada tarea.

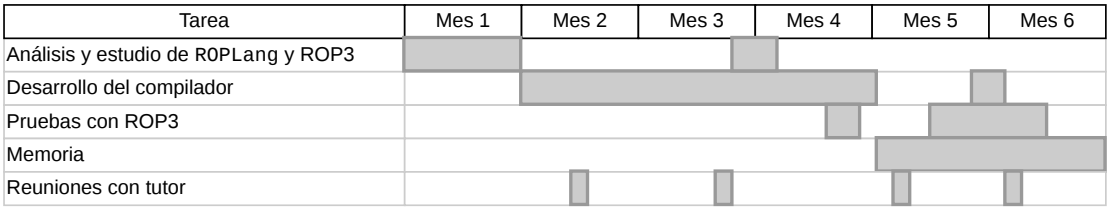

Figura A.1: Diagrama de Gantt del proyecto.

Tarea	**Esfuerzo (horas)**
Análisis y estudio de `ROPLang` y ROP3	19
Desarrollo del compilador	138
Pruebas con ROP3	25
Memoria	96
Reuniones tutor	4
Total	**282**

Tabla A.1: Esfuerzos dedicados a cada tarea.

Anexo B

Fichero con las funcionalidades implementadas

Este anexo recoge las funcionalidades implementadas en el sistema (Código B.1) y la traducción generada a su correspondiente código ROP (Código B.2). Para cada funcionalidad se muestra el código resultante, permitiendo relacionar las operaciones de alto nivel con las instrucciones de la representación intermedia generada por el compilador.

El objetivo de este listado es proporcionar una referencia detallada del proceso de traducción y facilitar la comprobación de que las funcionalidades implementadas se corresponden con las operaciones ROP esperadas. El Código B.2, incluido en este anexo, ha sido generado automáticamente a partir de las implementaciones desarrolladas durante el proyecto.

```
int main(int a) {
/////// Declarations
  int a, b;
  int a, b=0, c;

/////// Character tests
  char car = 'a';
  car = '\\';
  car = '\'';
  car = '\"';
  car = '\n';
  //car = 'ab'; // bad, not a character
  car = '\101'; // A
  car = '\x41'; // A

/////// Integer tests
  int thing = 1;
  int _thing = -1;
  //int 1thing = 1; // bad, invalid identifier
  thing = 0xa;
  thing = -0xa;
  //thing = 0xg;    // bad, invalid hex
  thing = 01;
  thing = -01;
  thing = 1u;
  thing = -1u;    // ok
  thing = 1lu;
```

```
  thing = +01ull;   // ok
  //thing = 01uuu;  // bad, too many 'u'
  //thing = 01lul;  // bad, no 'u' between two 'l's

/////// Real tests
  double a, b, c, d, e, f;
  a = 4.7;
  b = 4.;
  //c = 4;
  d = .7;
  e = 0.7;
  a = 4.7e1;
  b = 4.E-1;
  c = 4e+1;
  d = .7e1;
  e = 0.7e1;
  //f = .;  // bad, NaN
  //f = .e1;  // bad, NaN

/////// String tests
  //char* string = "string";
  char string = "string";
  string = "string \"";
  string = "string \\";
  string = "string \n";
  string = "Today's special is a pastrami sandwich on rye bread with \
  a potato knish and a cherry soda.";

/////// Operation tests. Supported operations: =, +, - (binary and
    unary), ^, &, |, ~ (unary), ==, !=, ! (unary), <, <=, >, >=, &&, ||,
     ++, --, *, /, %, <<, >>
/////// Also conditional statements: if, else, while, for
  a = b;
  a = 1 + 2;
  a = a + 1;
  a = 1 + b;
  a = b + c;
  a = a - b;
  a = 1 - 2;
  a = -a;
  a = -1;
  a = ~a;
  a = ~1;
  a = a == b;
  a = 4.7 == 4.7;
  a = a != b;
  a = 4.7 != 4.7;
  a = a + 2 - c;
  if (a == b) {
    a = 1;
  }
  if (a == b) {
    if (b != c) {
      a = 1;
    }
  }
  if (b == c) {
    if (a != b) {
```

```
      a = 1;
    } else {
      a = 2;
    }
  } else {
    a = 3;
  }
  a = !a;
  a = b < c;
  a = b > c;
  a = a <= b;
  a = a >= b;
  while (a <= b) {
    a = a + 1;
  }
  a = b && c;
  a = b && 0;
  a = b && 1;
  a = 0 && c;
  a = 1 && c;
  a = 0 && 0;
  a = 0 && 1;
  a = 1 && 0;
  a = 1 && 1;
  a = b || c;
  a = b || 0;
  a = b || 1;
  a = 0 || c;
  a = 1 || c;
  a = 0 || 0;
  a = 0 || 1;
  a = 1 || 0;
  a = 1 || 1;
  a = ++1;
  a = ++a;
  a = --1;
  a = --a;
  a = 1++;
  a = a++;
  a = 1--;
  a = a--;
  int i = 1, j = 2;
  for (i = 0; i < 10; i++) {
    a = a + i;
  }
  a = b * c;
  a = b * 2;
  a = 2 * c;
  a = 2 * 2;
  a = b / c;
  a = b / 2;
  a = 2 / c;
  a = 2 / 2;
  a = b % c;
  a = b % 2;
  a = 2 % c;
  a = 2 % 2;
  a = b << c;
```

```
142   a = b << 2;
143   a = 2 << c;
144   a = 2 << 2;
145   a = b >> c;
146   a = b >> 2;
147   a = 2 >> c;
148   a = 2 >> 2;
149   do {
150     a = a + 1;
151   } while (a < b);
152   a = ( i > j ? i : j );
153   a = ( i > j ? 1 : 2 );
154 }
```

Código B.1: Fichero `progress.c` (funcionalidades implementadas).

```
1
2
3
4  lc(regcar)
5  0x000000000000005c
6
7  lc(regcar)
8  0x0000000000000027
9
10 lc(regcar)
11 0x0000000000000022
12
13 lc(regcar)
14 0x000000000000000a
15
16 lc(regcar)
17 0x0000000000000041
18
19 lc(regcar)
20 0x0000000000000041
21
22
23
24 lc(regthing)
25 0x000000000000000a
26
27 lc(regthing)
28 0xfffffffffffffff6
29
30 lc(regthing)
31 0x0000000000000001
32
33 lc(regthing)
34 0xffffffffffffffff
35
36 lc(regthing)
37 0x0000000000000001
38
39 lc(regthing)
40 0xffffffffffffffff
41
42 lc(regthing)
```

```
0x0000000000000001

lc(regthing)
0x0000000000000001


lc(rega)
0x4012ccccccccccd

lc(regb)
0x4010000000000000

lc(regd)
0x3fe6666666666666

lc(rege)
0x3fe6666666666666

lc(rega)
0x4047800000000000

lc(regb)
0x3fd999999999999a

lc(regc)
0x4044000000000000

lc(regd)
0x401c000000000000

lc(rege)
0x401c000000000000


push 0x0000000000000000
push 0x2220676e69727473
mov(regstring, esp)

push 0x0000000000000000
push 0x5c20676e69727473
mov(regstring, esp)

push 0x0000000000000000
push 0x a20676e69727473
mov(regstring, esp)

push 0x0000000000000000
push 0x2e61646f73207972
push 0x7265686320612064
push 0x6e61206873696e6b
push 0x206f7461746f7020
push 0x61 9206874697720
push 0x6461657262206579
push 0x72206e6f20686369
push 0x77646e617320696d
push 0x6172747361702061
push 0x207369206c616963
push 0x6570732073277961
```

```
push 0x0000000000646f54
spa(5)
mov(regstring, esp)

mov(rega, regb)

lc(rega)
0x0000000000000003

mov(regResult0, rega)
lc(regRight)
0x0000000000000001
add(regResult0, regRight)
mov(rega, regResult0)

lc(regLeft)
0x0000000000000001
mov(regResult1, regLeft)
add(regResult1, regb)
mov(rega, regResult1)

mov(regResult2, regb)
add(regResult2, regc)
mov(rega, regResult2)

mov(regResult3, rega)
sub(regResult3, regb)
mov(rega, regResult3)

lc(rega)
0xffffffffffffffff

mov(regResult4, rega)
neg(regResult4)
mov(rega, regResult4)

lc(rega)
0xffffffffffffffff

mov(regResult5, rega)
not(regResult5)
mov(rega, regResult5)

lc(rega)
0xfffffffffffffffe

mov(regResult6, rega)
gcf-eqc(regResult6, regb)
mov(rega, regResult6)

lc(rega)
0x0000000000000001

mov(regResult7, rega)
gcf-eqc(regResult7, regb)
mov(regResult8, regResult7)
lc(regRight)
0x0000000000000001
```

```
xor(regResult8, regRight)
mov(rega, regResult8)

lc(rega)
0x0000000000000000

mov(regResult9, rega)
lc(regRight)
0x0000000000000002
add(regResult9, regRight)
mov(regResult10, regResult9)
sub(regResult10, regc)
mov(rega, regResult10)

mov(regResult11, rega)
gcf-eqc(regResult11, regb)
mov(regResult12, regResult11)
lsd(regResult12, 16)
spa(regResult12)
lc(rega)
0x0000000000000001


mov(regResult13, rega)
gcf-eqc(regResult13, regb)
mov(regResult17, regResult13)
lsd(regResult17, 88)
spa(regResult17)
mov(regResult14, regb)
gcf-eqc(regResult14, regc)
mov(regResult15, regResult14)
lc(regRight)
0x0000000000000001
xor(regResult15, regRight)
mov(regResult16, regResult15)
lsd(regResult16, 16)
spa(regResult16)
lc(rega)
0x0000000000000001



mov(regResult18, regb)
gcf-eqc(regResult18, regc)
mov(regResult22, regResult18)
lsd(regResult22, 120)
spa(regResult22)
mov(regResult19, rega)
gcf-eqc(regResult19, regb)
mov(regResult20, regResult19)
lc(regRight)
0x0000000000000001
xor(regResult20, regRight)
mov(regResult21, regResult20)
lsd(regResult21, 24)
spa(regResult21)
lc(rega)
0x0000000000000001
```

```
217
218 spa(, 16)
219 lc(rega)
220 0x0000000000000002
221
222
223 spa(, 16)
224 lc(rega)
225 0x0000000000000003
226
227
228 mov(regResult23, rega)
229 lc(regRight)
230 0x0000000000000001
231 xor(regResult23, regRight)
232 mov(rega, regResult23)
233
234 mov(regResult24, regb)
235 gcf-ltc(regResult24, regc)
236 mov(rega, regResult24)
237
238 mov(regResult25, regc)
239 gcf-ltc(regResult25, regb)
240 mov(rega, regResult25)
241
242 mov(regResult26, regb)
243 gcf-ltc(regResult26, rega)
244 mov(regResult27, regResult26)
245 lc(regRight)
246 0x0000000000000001
247 xor(regResult27, regRight)
248 mov(rega, regResult27)
249
250 mov(regResult28, rega)
251 gcf-ltc(regResult28, regb)
252 mov(regResult29, regResult28)
253 lc(regRight)
254 0x0000000000000001
255 xor(regResult29, regRight)
256 mov(rega, regResult29)
257
258 mov(regResult30, regb)
259 gcf-ltc(regResult30, rega)
260 mov(regResult31, regResult30)
261 lc(regRight)
262 0x0000000000000001
263 xor(regResult31, regRight)
264 mov(regResult33, regResult31)
265 lsd(regResult33, 48)
266 spa(regResult33)
267 mov(regResult32, rega)
268 lc(regRight)
269 0x0000000000000001
270 add(regResult32, regRight)
271 mov(rega, regResult32)
272
273 spa(, -112)
274
```

```
mov(regResult34, regb)
mov(regResult35, regResult34)
lc(regRight)
0x0000000000000000
gcf-eqc(regResult35, regRight)
mov(regResult36, regResult35)
lsd(regResult36, 8)
spa(regResult36)
mov(regResult34, regc)
mov(rega, regResult34)

mov(regResult37, regb)
mov(regResult38, regResult37)
lc(regRight)
0x0000000000000000
gcf-eqc(regResult38, regRight)
mov(regResult39, regResult38)
lsd(regResult39, 16)
spa(regResult39)
lc(regResult37)
0x0000000000000000
mov(rega, regResult37)

mov(regResult40, regb)
mov(regResult41, regResult40)
lc(regRight)
0x0000000000000000
gcf-eqc(regResult41, regRight)
mov(regResult42, regResult41)
lsd(regResult42, 16)
spa(regResult42)
lc(regResult40)
0x0000000000000001
mov(rega, regResult40)

lc(rega)
0x0000000000000000

mov(rega, regc)

lc(rega)
0x0000000000000000

lc(rega)
0x0000000000000000

lc(rega)
0x0000000000000000

lc(rega)
0x0000000000000001

mov(regResult43, regb)
mov(regResult44, regResult43)
lc(regRight)
0x0000000000000001
gcf-eqc(regResult44, regRight)
mov(regResult45, regResult44)
```

```
lsd(regResult45, 8)
spa(regResult45)
mov(regResult43, regc)
mov(rega, regResult43)

mov(regResult46, regb)
mov(regResult47, regResult46)
lc(regRight)
0x0000000000000001
gcf-eqc(regResult47, regRight)
mov(regResult48, regResult47)
lsd(regResult48, 16)
spa(regResult48)
lc(regResult46)
0x0000000000000000
mov(rega, regResult46)

mov(regResult49, regb)
mov(regResult50, regResult49)
lc(regRight)
0x0000000000000001
gcf-eqc(regResult50, regRight)
mov(regResult51, regResult50)
lsd(regResult51, 16)
spa(regResult51)
lc(regResult49)
0x0000000000000001
mov(rega, regResult49)

mov(rega, regc)

lc(rega)
0x0000000000000001

lc(rega)
0x0000000000000000

lc(rega)
0x0000000000000001

lc(rega)
0x0000000000000001

lc(rega)
0x0000000000000001

lc(rega)
0x0000000000000001

lc(regaux52)
0x0000000000000001
add(rega, regaux52)
mov(rega, rega)

lc(rega)
0x0000000000000001

lc(regaux53)
```

```
0x0000000000000001
sub(rega, regaux53)
mov(rega, rega)

lc(rega)
0x0000000000000001

mov(regResult54, rega)
lc(regaux55)
0x0000000000000001
add(rega, regaux55)
mov(rega, regResult54)

lc(rega)
0x0000000000000001

mov(regResult56, rega)
lc(regaux57)
0x0000000000000001
sub(rega, regaux57)
mov(rega, regResult56)


lc(regi)
0x0000000000000000
mov(regResult58, regi)
lc(regRight)
0x000000000000000a
gcf-ltc(regResult58, regRight)
mov(regResult62, regResult58)
lsd(regResult62, 64)
spa(regResult62)
mov(regResult61, rega)
add(regResult61, regi)
mov(rega, regResult61)

mov(regResult59, regi)
lc(regaux60)
0x0000000000000001
add(regi, regaux60)
spa(, -112)

mov(regResult63, regb)
lc(regiMul)
0x0000000000000000
mov(regResult64, regiMul)
gcf-ltc(regResult64, regc)
mov(regResult66, regResult64)
lsd(regResult66, 40)
spa(regResult66)
add(regResult63, regb)
lc(regaux65)
0x0000000000000001
add(regiMul, regaux65)
spa(, -72)
mov(rega, regResult63)

mov(regResult67, regb)
```

```
lc(regRight)
0x0000000000000002
lc(regiMul)
0x0000000000000000
mov(regResult68, regiMul)
lc(regRight)
0x0000000000000002
gcf-ltc(regResult68, regRight)
mov(regResult70, regResult68)
lsd(regResult70, 40)
spa(regResult70)
add(regResult67, regb)
lc(regaux69)
0x0000000000000001
add(regiMul, regaux69)
spa(, -88)
mov(rega, regResult67)

lc(regLeft)
0x0000000000000002
mov(regResult71, regLeft)
lc(regiMul)
0x0000000000000000
mov(regResult72, regiMul)
gcf-ltc(regResult72, regc)
mov(regResult74, regResult72)
lsd(regResult74, 40)
spa(regResult74)
add(regResult71, regLeft)
lc(regaux73)
0x0000000000000001
add(regiMul, regaux73)
spa(, -72)
mov(rega, regResult71)

lc(rega)
0x0000000000000004

mov(regResult75, regb)
lc(regiMul)
0x0000000000000000
lc(regzero)
0x0000000000000000
mov(regResult76, regResult75)
gcf-ltc(regResult76, regzero)
mov(regResult77, regResult76)
lc(regRight)
0x0000000000000001
xor(regResult77, regRight)
mov(regResult79, regResult77)
lsd(regResult79, 40)
spa(regResult79)
sub(regResult75, regc)
lc(regaux78)
0x0000000000000001
add(regiMul, regaux78)
spa(, -104)
mov(regResult75, i)
```

```
mov(rega, regResult75)

mov(regResult80, regb)
lc(regRight)
0x0000000000000002
lc(regiMul)
0x0000000000000000
lc(regzero)
0x0000000000000000
mov(regResult81, regResult80)
gcf-ltc(regResult81, regzero)
mov(regResult82, regResult81)
lc(regRight)
0x0000000000000001
xor(regResult82, regRight)
mov(regResult84, regResult82)
lsd(regResult84, 40)
spa(regResult84)
sub(regResult80, regRight)
lc(regaux83)
0x0000000000000001
add(regiMul, regaux83)
spa(, -104)
mov(regResult80, i)
mov(rega, regResult80)

lc(regLeft)
0x0000000000000002
mov(regResult85, regLeft)
lc(regiMul)
0x0000000000000000
lc(regzero)
0x0000000000000000
mov(regResult86, regResult85)
gcf-ltc(regResult86, regzero)
mov(regResult87, regResult86)
lc(regRight)
0x0000000000000001
xor(regResult87, regRight)
mov(regResult89, regResult87)
lsd(regResult89, 40)
spa(regResult89)
sub(regResult85, regc)
lc(regaux88)
0x0000000000000001
add(regiMul, regaux88)
spa(, -104)
mov(regResult85, i)
mov(rega, regResult85)

lc(rega)
0x0000000000000001

mov(regResult90, regb)
lc(regiMul)
0x0000000000000000
lc(regzero)
0x0000000000000000
```

```
mov(regResult91, regResult90)
gcf-ltc(regResult91, regzero)
mov(regResult92, regResult91)
lc(regRight)
0x0000000000000001
xor(regResult92, regRight)
mov(regResult94, regResult92)
lsd(regResult94, 40)
spa(regResult94)
sub(regResult90, regc)
lc(regaux93)
0x0000000000000001
add(regiMul, regaux93)
spa(, -104)
mov(rega, regResult90)

mov(regResult95, regb)
lc(regRight)
0x0000000000000002
lc(regiMul)
0x0000000000000000
lc(regzero)
0x0000000000000000
mov(regResult96, regResult95)
gcf-ltc(regResult96, regzero)
mov(regResult97, regResult96)
lc(regRight)
0x0000000000000001
xor(regResult97, regRight)
mov(regResult99, regResult97)
lsd(regResult99, 40)
spa(regResult99)
sub(regResult95, regRight)
lc(regaux98)
0x0000000000000001
add(regiMul, regaux98)
spa(, -104)
mov(rega, regResult95)

lc(regLeft)
0x0000000000000002
mov(regResult100, regLeft)
lc(regiMul)
0x0000000000000000
lc(regzero)
0x0000000000000000
mov(regResult101, regResult100)
gcf-ltc(regResult101, regzero)
mov(regResult102, regResult101)
lc(regRight)
0x0000000000000001
xor(regResult102, regRight)
mov(regResult104, regResult102)
lsd(regResult104, 40)
spa(regResult104)
sub(regResult100, regc)
lc(regaux103)
0x0000000000000001
```

```
623 add(regiMul, regaux103)
624 spa(, -104)
625 mov(rega, regResult100)
626
627 lc(rega)
628 0x0000000000000000
629
630 mov(regResult105, regb)
631 lc(regiMul)
632 0x0000000000000000
633 lc(regzero)
634 0x0000000000000000
635 mov(regResult106, regiMul)
636 gcf-ltc(regResult106, regc)
637 mov(regResult108, regResult106)
638 lsd(regResult108, 40)
639 spa(regResult108)
640 add(regResult105, regResult105)
641 lc(regaux107)
642 0x0000000000000001
643 add(regiMul, regaux107)
644 spa(, -72)
645 mov(rega, regResult105)
646
647 mov(regResult109, regb)
648 lc(regRight)
649 0x0000000000000002
650 lc(regiMul)
651 0x0000000000000000
652 lc(regzero)
653 0x0000000000000000
654 mov(regResult110, regiMul)
655 lc(regRight)
656 0x0000000000000002
657 gcf-ltc(regResult110, regRight)
658 mov(regResult112, regResult110)
659 lsd(regResult112, 40)
660 spa(regResult112)
661 add(regResult109, regResult109)
662 lc(regaux111)
663 0x0000000000000001
664 add(regiMul, regaux111)
665 spa(, -88)
666 mov(rega, regResult109)
667
668 lc(regLeft)
669 0x0000000000000002
670 mov(regResult113, regLeft)
671 lc(regiMul)
672 0x0000000000000000
673 lc(regzero)
674 0x0000000000000000
675 mov(regResult114, regiMul)
676 gcf-ltc(regResult114, regc)
677 mov(regResult116, regResult114)
678 lsd(regResult116, 40)
679 spa(regResult116)
680 add(regResult113, regResult113)
```

```
lc(regaux115)
0x0000000000000001
add(regiMul, regaux115)
spa(, -72)
mov(rega, regResult113)

lc(rega)
0x0000000000000008

mov(regResult117, regb)
lc(regiMul)
0x0000000000000000
lc(regzero)
0x0000000000000000
mov(regResult118, regResult117)
gcf-ltc(regResult118, regzero)
mov(regResult119, regResult118)
lc(regRight)
0x0000000000000001
xor(regResult119, regRight)
mov(regResult126, regResult119)
lsd(regResult126, 208)
spa(regResult126)
mov(regResult120, regResult117)
lc(regRight)
0x0000000000000002
lc(regiMul)
0x0000000000000000
lc(regzero)
0x0000000000000000
mov(regResult121, regResult120)
gcf-ltc(regResult121, regzero)
mov(regResult122, regResult121)
lc(regRight)
0x0000000000000001
xor(regResult122, regRight)
mov(regResult124, regResult122)
lsd(regResult124, 40)
spa(regResult124)
sub(regResult120, regRight)
lc(regaux123)
0x0000000000000001
add(regiMul, regaux123)
spa(, -104)
mov(regResult120, i)
lc(regaux125)
0x0000000000000001
add(regiMul, regaux125)
spa(, -272)
mov(rega, regResult117)

mov(regResult127, regb)
lc(regRight)
0x0000000000000002
lc(regiMul)
0x0000000000000000
lc(regzero)
0x0000000000000000
```

```
mov(regResult128, regResult127)
gcf-ltc(regResult128, regzero)
mov(regResult129, regResult128)
lc(regRight)
0x0000000000000001
xor(regResult129, regRight)
mov(regResult136, regResult129)
lsd(regResult136, 208)
spa(regResult136)
mov(regResult130, regResult127)
lc(regRight)
0x0000000000000002
lc(regiMul)
0x0000000000000000
lc(regzero)
0x0000000000000000
mov(regResult131, regResult130)
gcf-ltc(regResult131, regzero)
mov(regResult132, regResult131)
lc(regRight)
0x0000000000000001
xor(regResult132, regRight)
mov(regResult134, regResult132)
lsd(regResult134, 40)
spa(regResult134)
sub(regResult130, regRight)
lc(regaux133)
0x0000000000000001
add(regiMul, regaux133)
spa(, -104)
mov(regResult130, i)
lc(regaux135)
0x0000000000000001
add(regiMul, regaux135)
spa(, -272)
mov(rega, regResult127)

lc(regLeft)
0x0000000000000002
mov(regResult137, regLeft)
lc(regiMul)
0x0000000000000000
lc(regzero)
0x0000000000000000
mov(regResult138, regResult137)
gcf-ltc(regResult138, regzero)
mov(regResult139, regResult138)
lc(regRight)
0x0000000000000001
xor(regResult139, regRight)
mov(regResult146, regResult139)
lsd(regResult146, 208)
spa(regResult146)
mov(regResult140, regResult137)
lc(regRight)
0x0000000000000002
lc(regiMul)
0x0000000000000000
```

```
lc(regzero)
0x0000000000000000
mov(regResult141, regResult140)
gcf-ltc(regResult141, regzero)
mov(regResult142, regResult141)
lc(regRight)
0x0000000000000001
xor(regResult142, regRight)
mov(regResult144, regResult142)
lsd(regResult144, 40)
spa(regResult144)
sub(regResult140, regRight)
lc(regaux143)
0x0000000000000001
add(regiMul, regaux143)
spa(, -104)
mov(regResult140, i)
lc(regaux145)
0x0000000000000001
add(regiMul, regaux145)
spa(, -272)
mov(rega, regResult137)

lc(rega)
0x0000000000000000

mov(regResult147, rega)
lc(regRight)
0x0000000000000001
add(regResult147, regRight)
mov(rega, regResult147)

mov(regResult148, rega)
gcf-ltc(regResult148, regb)
mov(regResult149, regResult148)
lsd(regResult149, 72)
sps(regResult149)

mov(regResult150, regj)
gcf-ltc(regResult150, regi)
mov(regResult152, regResult150)
lsd(regResult152, 16)
spa(regResult152)
mov(regResult151, regi)
spa(, 8)
mov(regResult151, regj)
mov(rega, regResult151)

mov(regResult153, regj)
gcf-ltc(regResult153, regi)
mov(regResult155, regResult153)
lsd(regResult155, 24)
spa(regResult155)
lc(regResult154)
0x0000000000000001
spa(, 16)
lc(regResult154)
0x0000000000000002
```

```
855 mov(rega, regResult154)
```

Código B.2: Traducción generada por `progress.c`.