Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Trabajo Fin de Grado

Grado en Ingeniería Informática

Evaluación de la capacidad de ejecución de atacantes en sistemas Unix usando *Return Oriented Programming*

Evaluating the Execution Capabilities of Attackers on Unix Systems using Return Oriented Programming

Autor: Jaime Alconchel Gallego
Director: Ricardo Julio Rodríguez Fernández

Escuela de Ingeniería y Arquitectura
Junio 2026

Agradecimientos

A Ricardo por su labor tutorizando este trabajo.

Al Instituto de Investigación en Ingeniería de Aragón (I3A) por financiar parcialmente este trabajo a través de su Programa de Becas y Ayudas.

Declaración de uso responsable de IA

En este trabajo, se han utilizado los modelos Gemini 3.1 Pro, Claude Sonnet 4.6 y Claude Opus 4.8 para la generación de diagramas, la programación de algunos *scripts* y la depuración y detección de errores. Todo el contenido generado por esos modelos ha sido revisado y modificado por el autor, quien asume plena responsabilidad sobre el contenido íntegro de este trabajo, incluyendo aquellas partes en cuya elaboración se haya hecho uso de las herramientas mencionadas.

Resumen

La programación orientada al retorno (ROP, por sus siglas en inglés) es una técnica ofensiva de reutilización de código. Se basa en sobrescribir las direcciones de retorno en la pila para redirigir el flujo de ejecución a una serie de pequeños fragmentos (en inglés, *gadgets*) de *bytes* que interpretados aisladamente forman secuencias de instrucciones con una terminación de retorno propia de la arquitectura. La construcción de ataques ROP complejos es notoriamente complicada por la dificultad que supone encontrar secuencias de instrucciones con la finalidad deseada y sin demasiados efectos secundarios, logrando además una combinación de registros válida en los operandos de las instrucciones. Una herramienta que ayuda en la construcción de ataques ROP es `rop3`, desarrollada dentro del grupo de investigación DisCo de la Universidad de Zaragoza y dedicada a la localización de *gadgets* y a la construcción de secuencias de los mismos mediante un conjunto Turing-completo de pseudoinstrucciones. Sin embargo, presenta ciertas deficiencias en su versión actual. En este Trabajo de Fin de Grado (TFG), se expande este programa para medir la capacidad de ejecución mediante ROP en las principales bibliotecas de los sistemas operativos de la familia Unix. El software desarrollado y el banco de pruebas utilizado en este TFG se han publicado bajo la licencia GNU GPL v3.

Abstract

Return Oriented Programming (ROP) is an offensive code-reuse technique. It is based on overwriting return addresses on the stack to redirect the execution flow to a series of small byte fragments (known as gadgets) which, when interpreted in isolation, form instruction sequences with an architecture-native return termination. Building complex ROP attacks is notoriously difficult, due to the challenge of finding instruction sequences that serve the intended purpose without too many side effects, while also achieving a valid register combination in the instruction operands. One tool that aids in building ROP attacks is `rop3`, developed within the DisCo research group at the University of Zaragoza and dedicated to gadget discovery and to the construction of gadget chains through a Turing-complete set of pseudo-instructions. However, its current version has certain shortcomings. In this Bachelor's Thesis, this program is extended to measure ROP execution capability across the main libraries of Unix-family operating systems. Both the software developed and the test suite used in this thesis have been released under the GNU GPL v3 license.

Índice general

Índice de figuras

Índice de códigos

Índice de algoritmos

Capítulo 1

Introducción

1.1. Motivación

Las defensas software desarrolladas desde los primeros ataques informáticos han complicado los métodos ofensivos clásicos que únicamente aprovechaban una vulnerabilidad para inyectar instrucciones directamente ejecutables en un programa. Por ello, la reutilización de código, esto es, el aprovechamiento de instrucciones ya existentes en la zona de código de un programa para realizar ejecución arbitraria, constituye actualmente una técnica fundamental para diseñar ataques exitosos.

La programación orientada al retorno [1] (o ROP, del inglés *Return Oriented Programming*) es uno de los ataques de reutilización de código más comunes. Esta técnica se basa en el encadenamiento de secuencias de instrucciones (en inglés, *gadgets*) que finalizan en una instrucción de retorno y se ubican en las secciones ejecutables de un binario.

Localizar estas secuencias de fragmentos de instrucciones para construir ataques exitosos es una tarea compleja que a menudo se realiza manualmente con la ayuda de desensambladores u otros programas de análisis de aplicaciones. Para reducir la complejidad de esta tarea y visualizar la presencia de secuencias de instrucciones útiles en un binario en entornos Unix, en este Trabajo de Fin de Grado (TFG) se expande una herramienta ya existente dedicada al agrupamiento de *gadgets* en operaciones lógicas mediante el análisis estático (es decir, sin ejecutar el programa analizado) de ficheros binarios.

1.2. Objetivos, metodología y alcance

El objetivo de este TFG es extender[1] la herramienta `rop3` [2] a los sistemas operativos de la familia Unix y utilizarla para evaluar la capacidad de ejecución de un atacante mediante ROP en estos sistemas. Adicionalmente, se solucionan algunas de sus limitaciones actuales, con el fin de obtener resultados más exactos y representativos. Tanto `rop3` como los programas para obtener las bibliotecas y ejecutar las pruebas se publican bajo licencias de código abierto.

El alcance del trabajo se limita a las arquitecturas `x86` y `x86-64`, dejando la extensión de `rop3` a otras arquitecturas como trabajo futuro. El trabajo se limita también al análisis estático de binarios, de modo que no se consideran las técnicas de análisis dinámico [3].

[1] Extensión introducida en la solicitud de incorporación de cambios https://github.com/reverseame/rop3/pull/10

1.3. Estructura del documento

El presente trabajo se divide en seis capítulos y un anexo. El Capítulo 2 expone los conceptos necesarios para entender la herramienta desarrollada y las mediciones obtenidas. El Capítulo 3 se dedica específicamente a la descripción de `rop3` y de las extensiones que se le han realizado. El Capítulo 4 presenta la experimentación realizada y la discusión de resultados. El Capítulo 5 expone el estado del arte en ROP y el modo en el que `rop3` encaja en él. Finalmente, el Capítulo 6 expone las conclusiones globales del trabajo realizado. Adicionalmente, en el Anexo A se describe la distribución temporal del trabajo.

Capítulo 2

Conceptos previos

Este capítulo introduce los conceptos principales de la técnica de reutilización de código ROP y del lenguaje de abstracción de *gadgets* ROPLang, utilizado por `rop3`. Finalmente, se expone también el modelo de amenazas que se asume en los capítulos posteriores.

2.1. Programación orientada al retorno

Una fase esencial en la mayoría de los ciberataques es la ejecución arbitraria de código, es decir, la capacidad de que el atacante pueda ejecutar sus propias instrucciones en la máquina atacada. Para lograrlo, se parte de un programa vulnerable cuyo flujo de control puede ser redirigido a un código malicioso inyectado. Originalmente, bastaba con encontrar en el programa una vulnerabilidad de desbordamiento de *buffer* [4] que permitiera sobrescribir la dirección de retorno en la pila para dirigir el flujo de ejecución legítimo a una sección de código modificada por el atacante.

Sin embargo, con los años, la posibilidad de realizar directamente estos ataques prácticamente ha desaparecido, ya que los sistemas operativos modernos incluyen por defecto protecciones que evitan la ejecución de instrucciones situadas en zonas de memoria modificables por el usuario. Por ejemplo, DEP (del inglés *Data Execution Prevention*) o `W^X` (del inglés *write xor execute*) son las mitigaciones que imposibilitan la ejecución de código en secciones de datos en Windows [5] y OpenBSD [6], respectivamente.

Por tanto, si se pretende ejecutar instrucciones de forma malintencionada, estas deben estar en secciones ejecutables de memoria. De esta manera, si el atacante logra escribir en la pila, ya no puede simplemente sobrescribir la dirección de retorno con una dirección a la sección de datos, puesto que no es ejecutable.

En las arquitecturas `x86` y `x86-64`, al llamar a una función mediante la instrucción del ensamblador `call`, se almacena la dirección de retorno de la instrucción que sucede al `call` en la pila. Esto permite que al finalizar la función llamada se pueda reanudar el contexto desde el que se entró en ella, utilizando la instrucción del ensamblador `ret`, que carga en el puntero de instrucción (el que almacena la próxima instrucción a ejecutar por el procesador) la dirección de retorno almacenada en la pila. Este diseño, esencial para el funcionamiento de la arquitectura, es vulnerable a ataques que modifican arbitrariamente la pila, lo que lo convierte en una puerta de entrada idónea para pasar de la manipulación de datos a la ejecución de código.

Los primeros ataques en aprovechar esta vulnerabilidad solían sobrescribir la dirección de retorno con la de otra función, generalmente de la biblioteca estándar de C (abreviada `libc`) en sistemas Unix, por lo que se denominaron ataques *return-to-libc* [7, 8]. Sin

embargo, este enfoque resultaba muy limitado, ya que el atacante debía ceñirse a la funcionalidad concreta de la función que iba a ejecutar.

Partiendo de los ataques *return-to-libc* y aprovechando el diseño de los procesadores `x86`/`x86-64`, en [1] se propuso una idea clave: la dirección de retorno no tiene que apuntar al inicio de una función, sino que puede hacerlo a cualquier instrucción en el código, incluso si no está alineada, siempre que la sucedan unas pocas instrucciones más y, finalmente, otra instrucción `ret`. Cuando se afirma que la instrucción a la que se salta puede no estar alineada, esto es porque `x86` y `x86-64` son arquitecturas en las que las instrucciones tienen tamaño variable, por lo que existe la posibilidad de comenzar a ejecutar código desde cualquier *byte* de la zona de instrucciones. Este detalle hace que en un análisis exhaustivo de memoria afloren muchos *gadgets* con instrucciones "accidentales" que no forman parte de ninguna función, pero que pueden aprovecharse para un ataque de reutilización de código.

Como la instrucción `ret` toma la dirección almacenada en la pila y salta a ella, si el atacante logra modificarla (por ejemplo, con un desbordamiento de *buffer* [4]), puede controlar adónde salta el programa. La novedad de ROP es que, escribiendo cuidadosamente en la pila, se pueden encadenar varias direcciones de retorno: cada fragmento de código se ejecuta y, al llegar a la instrucción `ret`, salta a la siguiente dirección elegida. El resultado de este proceso es que el atacante es capaz de ejecutar una secuencia de instrucciones sin necesidad de inyectar nuevo código.

A esta técnica de explotación se la denominó programación orientada al retorno, por su abuso de la instrucción de retorno de la arquitectura, y a los pequeños fragmentos acabados en una instrucción `ret`, *gadgets*. Por su parte, a las secuencias de *gadgets* se las llama cadenas o *ROPchains*. La Figura 2.1 muestra un ejemplo de ejecución de una *ROPchain* que carga dos valores de la pila en `rax` y `rcx` y los suma.

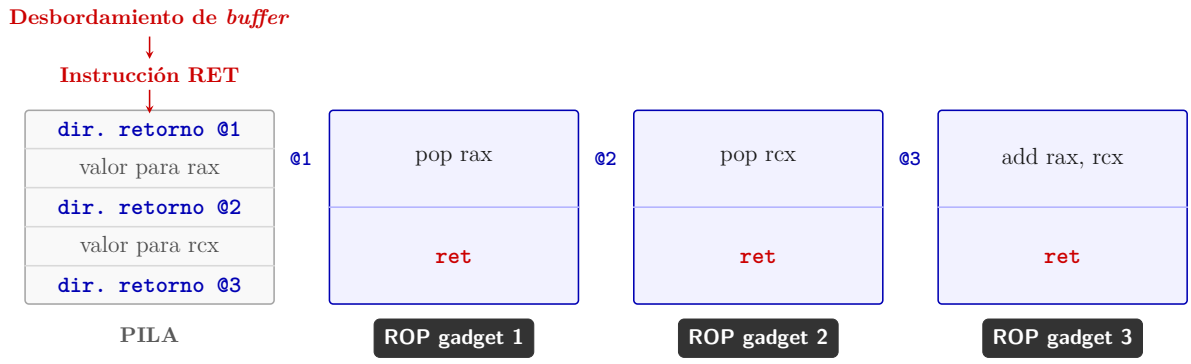

Figura 2.1: Ejemplo de ejecución de una *ROPchain*

2.2. El lenguaje ROPLang

Si bien es cierto que cada ROP *gadget* puede tener funcionalidades muy diversas que afectan de distinto modo a los registros de la arquitectura, al tratarse de secuencias breves de instrucciones, se pueden detectar patrones en muchos de ellos. Por ejemplo, los *gadgets* «`add rax, rcx; ret`» y «`sub rbx, rdx; ret`» realizan operaciones aritméticas sobre distintos registros, mientras que el *gadget* «`mov rax, [rcx]; ret`» realiza una lectura de memoria. Esta categorización puede extenderse a muchos de los *gadgets* útiles presentes en un binario, lo que supone una abstracción potente y relevante.

La idea de dividir los distintos ROP *gadgets* `x86/x86-64` en pequeñas operaciones de un lenguaje Turing-completo aparecía ya en la publicación original sobre ROP [1] y en otras que la siguieron [9]. Posteriormente, implementaciones como [10] lograron ir más allá y definieron un lenguaje de alto nivel con soporte para funciones y una pila emulada en la sección de datos del programa. Aunque eran interesantes teóricamente, resultaban en la práctica limitadas a la hora de explotar binarios reales, pues presuponían unos *gadgets* y una configuración del binario muy específicos, además de requerir generalmente una entrada maliciosa (en inglés, *payload*) enorme.

El lenguaje ROPLANG [2], que se utilizará en este trabajo, supone una aproximación intermedia, definiendo operaciones más completas y potentes que las de [1], pero manteniendo un lenguaje similar al ensamblador, en el que el control de flujo se realiza manualmente con una serie de operaciones, y prescindiendo además de emular una pila de datos como en [10]. Por tanto, ROPLANG es un lenguaje definido directamente para facilitar la creación de cadenas eficaces en la explotación de binarios, eliminando la necesidad de buscar *gadgets* específicos para realizar una función común, pero evitando grandes abstracciones que resultarían contraproducentes al tratar de localizar fragmentos explotables en el código.

ROPLANG está definido únicamente para la arquitectura `x86`, si bien soporta directamente las extensiones de 64 bits. Sus operaciones pueden dividirse en los siguientes grupos:

- **Operaciones aritméticas:** `add`, `sub` y `neg`.
- **Operaciones lógicas:** `xor`, `and`, `or` y `not`.
- **Operaciones de asignación:** `mov` y `lc`.
- **Operaciones de referencia:** `ld` y `st`.

Cada operación básica de ROPLANG corresponde a uno o varios ROP *gadgets* estándar, es decir, a una secuencia de instrucciones presentes en la sección de código del programa que finalizan en una instrucción `ret`. Las operaciones básicas del lenguaje se definen en la Tabla 2.1.

Operación	ROP *gadgets*	Descripción

`add(dst, src)`	`add dst, src`
	`clc`
	`adc dst, src`
		Suma a `dst` el valor de `src`

`adc(dst, src)`	`adc dst, src`	Suma a `dst` el valor de `src` más el acarreo

`sub(dst, src)`	`sub dst, src`
	`clc`
	`sbb dst, src`
		Resta a `dst` el valor de `src`

`neg(dst)`	`neg dst`	Cambia el valor de `dst` a su complemento a 2
`and(dst, src)`	`and dst, src`	dst := dst $\wedge$ src
`xor(dst, src)`	`xor dst, src`	dst := dst $\oplus$ src
`or(dst, src)`	`or dst, src`	dst := dst $\vee$ src

`not(dst)`	`not dst`	dst := ¬dst
`mov(dst, src)`	`mov dst, src`	Mueve a `dst` el valor de `src`
	`push src` `pop dst`	
	`xchg dst, src`	
	`xor dst, dst` `add dst, src`	

	`xor dst, dst`
	`not dst`
	`and dst, src`	

	`clc` `cmovnc dst, src`	
	`stc` `cmovc dst, src`	
`lc(dst)`	`pop dst`	Almacena en `dst` el valor actual en la pila
`ld(dst, src)`	`mov dst, [src]`	Almacena en `dst` el valor en memoria apuntado por `src`
`st(dst, src)`	`mov [dst], src`	Almacena en la dirección apuntada por `dst` el valor de `src`
`leave`	`leave`	Operación necesaria para construir operaciones complejas

Tabla 2.1: Operaciones básicas de ROPLANG (se omite la instrucción `ret` al final de cada *gadget*)

Una vez definidas las operaciones básicas del lenguaje, es posible construir operaciones compuestas a partir de ellas para lograr ejecución condicional y otras acciones avanzadas. Las operaciones compuestas se describen en la Tabla 2.2.

Operación	Ops. ROPLANG	Descripción
`eqc(dst, src)`	`sub(dst, src)` `neg(dst)`	Guarda en la bandera de acarreo si dst ≠ src
`ltc(dst, src)`	`sub(dst, src)`	Guarda en la bandera de acarreo si dst < src

`gcf(dst, eqc(REG1, REG2))`	`xor(dst, dst)`
	`eqc(REG1, REG2)`
	`adc(dst, dst)`
		Guarda en `dst` el resultado de aplicar el operador condicional `eqc` a `REG1`, `REG2`

`spa(src)`	`add(SP, src)`	Suma al puntero de pila el valor almacenado en `src`
`sps(src)`	`sub(SP, src)`	Resta al puntero de pila el valor almacenado en `src`

`gsp(dst)`	`mov(dst, SP)`	Guarda el valor del puntero de pila en `dst`
`jmp(src)`	`mov(BP, src)` `leave`	Realiza un salto en el contexto de ROP a `src`
`jmp-rel(src)`	`spa(src)`	Realiza un salto relativo en el contexto de ROP con desplazamiento `src`

Tabla 2.2: Operaciones compuestas de ROPLang

Para comprender algunas de estas instrucciones es necesario conocer el funcionamiento de la bandera de acarreo en `x86/x86-64`. La bandera de acarreo es un bit del registro de estado del procesador que vale 1 si el resultado de una suma de enteros sin signo excede el tamaño del registro o si el resultado de una resta de enteros sin signo es menor que 0. Este último caso es el que explota ROPLang para implementar operaciones de comparación y saltos condicionales (considerar que `neg(rax)` es equivalente a hacer la resta 0 - `rax`).

Se debe tener en cuenta que, en la búsqueda de *gadgets*, se dan muchos casos en los que sus primeras instrucciones se corresponden con las de una operación ROPLang definida, pero que incluyen instrucciones ensamblador adicionales antes de alcanzar el `ret`. Si alguna de estas instrucciones modifica el valor de otros registros o de la memoria, se dice que el *gadget* tiene efectos secundarios. Por ejemplo, «`add rax, rcx; pop rbp; ret`» corresponde a una operación ROPLang `add(rax, rcx)`, pero modifica también el valor de `rbp` como efecto secundario. Idealmente, estos *gadgets* no se tendrían en cuenta, pero en un escenario de *gadgets* escasos, descartarlos directamente puede eliminar las posibilidades de llevar a cabo un ataque ROP exitoso. En definitiva, estos *gadgets* deben considerarse, pero al encadenarlos hay que verificar que los efectos secundarios que se acumulan no corrompen acciones anteriores en otros registros.

2.3. Modelo de amenazas

Siguiendo las buenas prácticas en trabajos académicos sobre ciberseguridad ofensiva, se define a continuación un modelo de amenazas, esto es, un conjunto de supuestos sobre la capacidad del atacante que permiten definir pruebas realistas y extraer conclusiones con valor real. En este TFG se asume:

1. Que el atacante es capaz de manipular de forma maliciosa la pila para introducir la *ROPchain*, sobrescribiendo las direcciones de retorno necesarias.

2. Que el atacante es capaz de escribir en una sección de memoria no necesariamente ejecutable (de ser ejecutable, ROP resultaría innecesario) y de dimensión suficiente.

3. Que el atacante conoce las direcciones absolutas de los ROP *gadgets* y de la zona de memoria en la que puede escribir.

Estos supuestos no son exageradamente optimistas. Muchos ataques ROP exigen al menos los dos primeros y requieren además de algún tipo de fuga de información que permita conocer o aproximar las direcciones de los ROP *gadgets* y de otras zonas de memoria, superando defensas software modernas como ASLR [11], que aleatoriza la disposición en memoria del código y los datos. A pesar de todo, ROP sólo precisa estrictamente la

capacidad de modificar las direcciones de retorno en la pila, y muchas conclusiones de este trabajo acerca de la capacidad de ejecución del atacante en relación al número de *gadgets* pueden generalizarse. No obstante, todas las *ROPchains* diseñadas asumen este modelo de amenazas.

Capítulo 3

Extensión de rop3

Este capítulo expone el funcionamiento de la herramienta rop3 y las extensiones que se le han realizado. Para obtener las nuevas mediciones, se extiende el soporte binario de rop3 a los principales sistemas Unix. Adicionalmente, se implementan algunas funcionalidades definidas como trabajo futuro en [2]. El código fuente de rop3, incluyendo las extensiones, está disponible en [12]. Finalmente, se redefine la prueba de completitud de Turing del trabajo original sobre ROPLang, para poder probarla experimentalmente en binarios reales.

3.1. Descripción de rop3

El programa rop3 es una herramienta escrita en Python 3 dedicada a la localización de ROP *gadgets* agrupados en operaciones ROPLang en uno o más binarios del sistema operativo Windows. A nivel técnico, rop3 se aprovecha del *framework* de desensamblado Capstone [13], que incorpora funcionalidades como el análisis de los registros leídos y escritos por una o varias instrucciones o la conversión de secuencias de *bytes* de instrucciones en diccionarios de Python fáciles de analizar.

La herramienta rop3 localiza ROP *gadgets* compatibles con operaciones del lenguaje ROPLang del siguiente modo: para cada operación, encuentra aquellos que realizan la acción deseada sobre un conjunto de registros. Estos pueden ser abstractos, como en «`add(REG1, REG2)`», si el usuario desea que sea la propia herramienta la que seleccione los registros apropiados; explícitos, como en «`add(rax, rdx)`», si se requiere que la operación emplee unos registros concretos; o incluso ser sustituidos por inmediatos, como en «`add(REG1, 8)`». El algoritmo de búsqueda de *gadgets* se basa en localizar terminaciones válidas de *gadget* y leer las instrucciones hacia atrás en el código para tratar de construir un *gadget* válido. El número de *bytes* que se lee hacia atrás se fija mediante un parámetro de entrada (profundidad). El funcionamiento del algoritmo que extrae *gadgets* de un binario se describe en el Algoritmo 1.

El programa desarrollado también es capaz de encadenar operaciones ROPLang entre sí, buscando en el espacio de los *gadgets* y sus operandos (los registros, inmediatos o direcciones de memoria que leen o modifican) combinaciones válidas. Por ejemplo, para encadenar la secuencia de operaciones ROPLang «`lc(REG1); lc(REG2); add(REG1, REG2)`» en el espacio de *gadgets* formado por «`pop rax; ret`», «`pop rcx; ret`», «`pop rbp; ret`» y «`add rbp, rax; ret`», rop3 calcula que la única combinación válida es «`REG1 = rbp`» y «`REG2 = rax`», generando la cadena «`pop rbp; ret; pop rax; ret; add rbp, rax; ret`».

Para obtener combinaciones válidas de registros, rop3 indexa los gadgets de cada

Algoritmo 1: BUSCARGADGETS

Entrada: *bin*: archivo binario; *profundidad* de búsqueda
Salida: Lista de *gadgets* encontrados en el binario

```
1  G ← [ ]
2  T ← terminaciones de gadgets                          // ret, retf, ...
3  para cada terminación t ∈ T hacer
4      para cada sección ejecutable s ∈ bin hacer
5          para cada potencial gadget ref de t.bytes en s.opcodes hacer
6              para d ← t.tamaño a profundidad hacer
7                  dir ← dirección base de s + ref − d
8                  si EsDirecciónVálida(dir) entonces
9                      bytes ← s.opcodes[ ref − d : ref ]
10                     insns ← Desensamblar(bytes, dir)
11                     si EsGadgetVálido(insns) entonces
12                         G.append(Gadget(dir, insns, bytes))
13 devolver G
```

operación ROPLANG por sus registros fuente, destino o ambos, y luego utiliza un algoritmo de búsqueda por retroceso para lograr una asignación válida de registros concretos a los registros abstractos. Para construir *ROPchains*, la herramienta sólo considera los *gadgets* en función de sus registros fuente y destino, sin tener en cuenta otras acciones que puedan realizar.

El procedimiento descrito anteriormente se concreta en dos algoritmos. El Algoritmo 2 genera un estado inicial con todas las asignaciones posibles de registros y los gadgets que las realizan. El Algoritmo 3 trata de generar combinaciones coherentes de registros a partir de las asignaciones posibles, utilizando un algoritmo de búsqueda por retroceso.

En la versión base de `rop3`, los tres algoritmos ya expuestos son suficientes para generar *ROPchains*. Una vez que se obtenían todas las combinaciones posibles de registros, se asignaban a cada combinación posible sus *gadgets* concretos. La construcción de cadenas en `rop3` operaba por tanto a nivel de registros y no de instrucciones, reduciendo considerablemente la complejidad computacional que supondría una búsqueda en profundidad sobre todo el espacio de *gadgets*.

3.2. Cambios introducidos

Antes de las modificaciones en `rop3` llevadas a cabo en este trabajo la herramienta presentaba las siguientes limitaciones: (i) sólo podía analizar binarios de Windows; (ii) consideraba como válidos *gadgets* con saltos intermedios que nunca alcanzaban el `ret`, como en «`pop rax; jmp 0xbeba; ret`»; (iii) la representación de los *gadgets* como diccionarios dificultaba el desarrollo y la introducción de nuevas funcionalidades; (iv) las constantes de la arquitectura `x86`/`x86-64` se encontraban dispersas por el código de la herramienta, impidiendo portar `rop3` a otras arquitecturas; (v) las operaciones ROPLANG necesarias para realizar ejecución condicional eran inexistentes en la mayoría de los binarios; y (vi) no se consideraban los efectos secundarios en la construcción de cadenas.

Algoritmo 2: EstadoInicial

Entrada: *gadgets*: todos los gadgets del binario; *cadena*: pasos (op, dst, src)
Salida: *valores*: asignaciones posibles de registros, $gadgets_{op}$: *gadgets* agrupados por su posición en la cadena, *pares*: lista de registros abstractos → concretos que aparecen juntos en una instrucción

```
1  valores ← tabla vacía registro abstracto → conjunto de concretos
2  gadgets_op ← lista vacía
3  pares ← lista vacía
4  para cada paso ∈ cadena hacer
5      G ← FiltrarGadgets(paso, gadgets)   // gadgets que implementan esta
         operación
6      si G está vacío entonces
7          devolver lista vacía
8      añadir G a gadgets_op
9      (d, s) ← (paso.dst, paso.src)
10     si d y s son ambos abstractos entonces
11         P ← conjunto de (g.dst, g.src) para todo g ∈ G con destino y fuente
             concretos
12         añadir (d, s, P) a pares
13         vals_d ← los destinos que aparecen en P
14         vals_s ← las fuentes que aparecen en P
15     si no
16         añadir "sin restricción" a pares
17         vals_d ← {g.dst : g ∈ G} si d es abstracto, en otro caso vacío
18         vals_s ← {g.src : g ∈ G} si s es abstracto, en otro caso vacío
19     para cada (clave, vals) ∈ {(d, vals_d), (s, vals_s)} hacer
20         si clave no es abstracto entonces
21             continuar
22         si clave ya está en valores entonces
23             valores[clave] ← valores[clave] ∩ vals
24         si no
25             valores[clave] ← vals
26 devolver (valores, gadgets_op, pares)
```

Algoritmo 3: GenerarCombinaciones

Entrada: *valores*: lista de pares (registro abstracto, registros concretos posibles);
pares: por cada paso, "sin restricción" o $(clave_{dst}, clave_{src}, P)$
Salida: *resultados*: lista de asignaciones {abstracto → concreto}

```
1  Función Backtrack(i, actual, usados):
2      si i = |valores| entonces
3          si actual satisface alguna combinación de pares entonces
4              añadir una copia de actual a resultados
5          devolver
6      (clave, posibles) ← valores[i]
7      para cada v ∈ posibles hacer
8          si v ∈ usados entonces
9              continuar
10         asignar actual[clave] ← v
11             marcar v como usado
12         Backtrack(i + 1, actual, usados)
13         deshacer la asignación de clave
14             desmarcar v
15 resultados ← lista vacía
16 Backtrack(0, asignación vacía, conjunto vacío)
17 devolver resultados
```

3.2.1. Soporte de nuevos formatos y rediseño de la arquitectura

Los cuatro primeros puntos no implican cambios conceptuales en `rop3`, y se resuelven de distintas formas. Concretamente:

(i) Soporte de binarios macOS. Para solucionar la primera limitación, se implementa soporte para binarios de macOS (`Mach-O`) utilizando la biblioteca `macholib` [14]. El soporte para Linux/BSD (binarios `ELF`) se había implementado en una versión posterior a la publicación de [2], utilizando la biblioteca `pyelftools` [15].

(ii) Descarte de *gadgets* con saltos intermedios. Para solucionar la segunda limitación, se añade a la verificación de validez del *gadget* presente en el Algoritmo 1 la condición de que el *gadget* no contenga saltos incondicionales que lo invaliden siempre (instrucciones ensamblador `call` y `jmp`). Sin embargo, se mantienen opcionalmente los *gadgets* con saltos condicionales, como «`mov rax, rdx; jne 0xcafe; ret`», ya que pueden resultar funcionales dependiendo del contexto.

(iii) y (iv) Rediseño de la arquitectura. A nivel arquitectural, en la nueva versión de `rop3` los *gadgets* han dejado de ser simples diccionarios de Python 3 y se han convertido en una clase de datos del lenguaje, permitiendo definir heurísticas, órdenes y filtros con más facilidad y coherencia. Además, todas las constantes de Capstone vinculadas a la arquitectura `x86` se han eliminado de los módulos de la aplicación, encapsulándolas en

una clase abstracta `Architecture` que está definida actualmente para las arquitecturas `x86` y `x86-64`. La nueva arquitectura de `rop3` puede visualizarse en la Figura 3.1.

3.2.2. Nuevas operaciones de ejecución condicional

La ejecución condicional en ROP es una mecánica compleja que admite múltiples implementaciones, cada una de las cuales presenta ventajas y desventajas. Siguiendo una interpretación teórica de ROP en la que el puntero de pila, al controlar la dirección a la que salta la próxima instrucción de retorno, reemplaza funcionalmente al puntero de instrucción, la forma idónea de implementar condicionalidad es escribir valores indirectos (almacenados en registros o en memoria) en el puntero de pila. La aproximación empleada en [2] es combinar las operaciones de comparación de ROPLang (`ltc` y `eqc`), la de extracción de la bandera de acarreo `gcf` y la de salto relativo `jmp-rel`.

Sin embargo, este enfoque presenta un problema: los *gadgets* que realizan la operación de salto relativo `jmp-rel` con un valor indirecto como fuente son muy escasos (casi todas las operaciones `spa` son del tipo «`add rsp, #imm`», es decir, suman valores constantes). Por tanto, resulta interesante buscar otras formas de implementar saltos condicionales.

Las instrucciones ensamblador `mov` o `xchg` con el puntero de pila en uno de los operandos son igual de escasas que las de salto relativo y, de existir, en casi todos los casos el otro operando es el puntero de marco (el que se emplea para apuntar a la base del marco de pila de una función). No obstante, existe un *gadget* prevalente en binarios `x86`/`x86-64` que permite escribir un valor en el puntero de pila desde el puntero de marco: «`leave; ret`». Si fuera posible escribir valores condicionales en el puntero de marco, se lograría el objetivo de manipular arbitrariamente el valor del puntero de pila. De hecho, existen múltiples formas de almacenar valores en este registro, ya que por ejemplo los ROP *gadgets* del tipo «`pop ebp; ret`» son bastante comunes, donde `ebp` es el puntero de marco en 32 bits.

Como parte de la extensión de la herramienta, se ha renombrado la antigua operación `jmp` como `jmp-rel` y se ha redefinido `jmp`, que ahora trabaja directamente con direcciones de salto absolutas. Esta modificación aparece ya reflejada en la Tabla 2.2.

En el Código 3.1 se muestra un ejemplo de cómo utilizar ROPLang y la nueva operación de salto para realizar ejecución condicional de forma realista. En este ejemplo, se utiliza la bandera de acarreo para generar dos máscaras de bits (todo ceros o todo unos) que se aplican a sendas direcciones de salto condicionales.

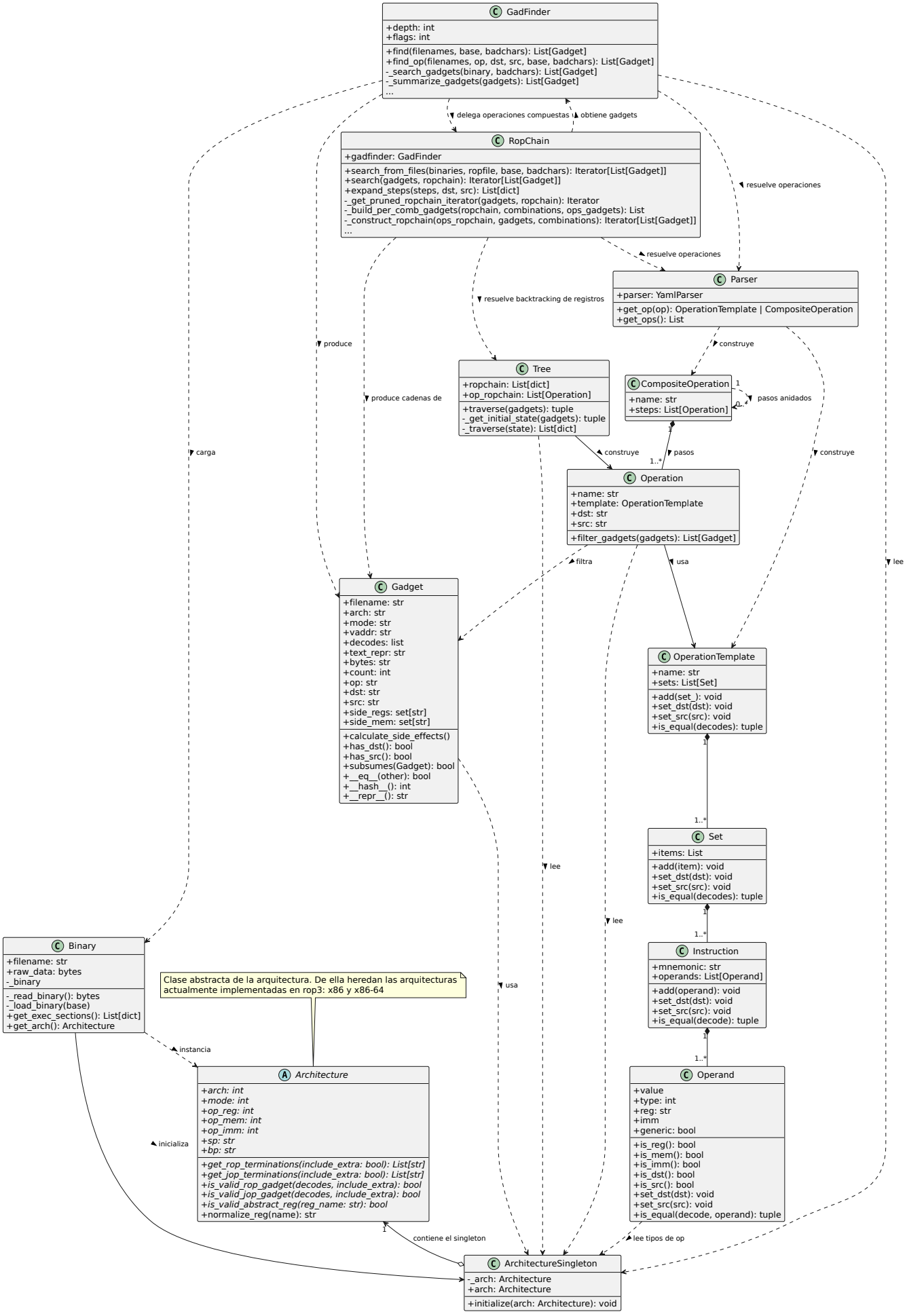

Figura 3.1: Diagrama de clases de `rop3`

Código 3.1: Ejemplo de ejecución condicional en la nueva versión de `rop3` (32 bits)

```
lc(reg1) ; reg1 = a
lc(reg2) ; reg2 = b
lc(reg3) ; reg3 = @jump_1
lc(reg4) ; reg4 = @jump_2
; if (a == b)
;     goto jump_1;
; else
;     goto jump_2;
sub(reg1, reg2)
neg(reg1)
lc(reg1) ; reg1 = 0
adc(reg1, reg1)
lc(reg5) ; reg5 = 0xFFFFFFFF
lc(reg6) ; reg6 = 0x00000000
add(reg5, reg1) ; reg5 = 0 if reg1 == 1 else 0xFFFFFFFF
sub(reg6, reg1) ; reg6 = 0 if reg1 == 0 else 0xFFFFFFFF
and(reg3, reg5)
and(reg4, reg6)
add(reg3, reg4)
; reg3 = (a == b) AND @jump_1 + (a != b) AND @jump_2
jmp(reg3)
```

3.2.3. Rastreo de efectos secundarios

Otra limitación de `rop3` es que no considera los efectos secundarios de los ROP *gadgets* al construir cadenas. Se recuerda que se llama efecto secundario a una modificación en los registros o en la memoria que realiza un *gadget* sin ser esta su finalidad propia. La versión previa de la herramienta `rop3` no considera que en las arquitecturas `x86` y `x86-64` varios alias corresponden al mismo registro lógico (a estos registros se los llamará registros solapados). Por ejemplo, las instrucciones ensamblador «**`mov rax, rbp`**» y «**`mov eax, ebp`**» actúan sobre los mismos registros lógicos del procesador, aunque no operan igual a nivel de bits: la primera instrucción copia los 64 bits de **`rbp`** a **`rax`**, mientras que la segunda sólo lo hace con los 32 bits de menor peso y anula los superiores. No tener en cuenta esta particularidad de la arquitectura no supone un problema si no se resuelven los efectos secundarios, ya que al construir combinaciones de registros concretos a partir de los abstractos, debe accederse a estos en el mismo modo de bits para evitar problemas de pérdida de información. Sin embargo, si se desea llevar la cuenta de los efectos secundarios que tienen los ROP *gadgets*, es necesario procesar estas equivalencias.

En síntesis, `rop3` trata los registros solapados como si fueran independientes al resolver sus equivalentes abstractos en las cadenas (por ejemplo, `REG1` puede asignarse a **`rax`** o a **`eax`** en la operación `lc(REG1)`, pero no a los dos al mismo tiempo); mientras que si un *gadget* modifica como efecto secundario cualquier registro, todos los registros solapados se consideran afectados. Por ejemplo, si se altera como efecto secundario el valor de **`eax`**, se marcan también como afectados **`rax`**, **`al`**, etc.

Una vez introducidos los efectos secundarios, se lleva a cabo una formalización de los *gadgets*:

Definición 3.2.1 (Gadget de ROPLang)**.** Sea $\mathcal{I}$ el conjunto de operaciones del lenguaje ROPLang, sea $\mathcal{R}$ el conjunto de registros de la arquitectura, sea $\mathcal{R}_n$ el conjunto de los

registros normalizados (los de 32 o 64 bits, dependiendo de la arquitectura) y sea $\mathcal{V} \subset \mathbb{Z}$ el conjunto de valores inmediatos.

Un *gadget* que realiza una operación ROPLang queda definido como la tupla:

$$\langle \tau,\ dst,\ src,\ \Delta \rangle,$$

donde:

- $\tau \in \mathcal{I}$ es la operación de ROPLang que realiza el *gadget*;
- $dst \in \mathcal{R} \cup \{\bot\}$ y $src \in \mathcal{R} \cup \mathcal{V} \cup \{\bot\}$ son los operandos destino y fuente, donde $\bot$ denota la ausencia de operando;
- $\Delta \subseteq \mathcal{R}_n$ es el conjunto de registros alterados como efecto secundario, normalizados de modo que un registro y sus versiones de menor tamaño se consideran equivalentes.

Definición 3.2.2 (Subsunción de *gadgets*)**.** Sobre el conjunto de *gadgets* se define la relación de subsunción $\preceq$. Dados dos *gadgets* $g = \langle \tau,\ dst,\ src,\ \Delta \rangle$ y $g' = \langle \tau',\ dst',\ src',\ \Delta' \rangle$, se dice que g' subsume a g, y se escribe $g' \preceq g$, si y solo si realizan la misma operación sobre los mismos operandos y los efectos secundarios de g' están contenidos en los de g:

$$g' \preceq g \iff \tau = \tau' \wedge dst = dst' \wedge src = src' \wedge \Delta' \subseteq \Delta.$$

Aplicando la relación al conjunto de *gadgets* que realizan la misma operación sobre los mismos operandos fuente y destino, la subsunción induce un orden parcial sobre los *gadgets* con la misma tupla del que se aprovecharán los algoritmos diseñados. La subsunción es fundamental a la hora de purgar el espacio de búsqueda de *gadgets* en el Algoritmo 4, ya que al sustituir los *gadgets* con más efectos secundarios por otros equivalentes elimina los más problemáticos, sin reducir el número de posibles combinaciones.

Se decide añadir una restricción opcional a la creación de *ROPchains*: que los registros abstractos se resuelvan siempre con aquellos que utilicen todos los bits de la arquitectura. Por ejemplo, en el caso de 64 bits, `REG1` se podría asignar a `rax` o `rbp`, pero no a sus equivalentes de 32 bits como `eax` o `ebp` o de 8 bits como `al`. Esto evita generar cadenas que utilizan las versiones inferiores de los registros de la arquitectura, limitando en ocasiones la capacidad de ejecución de un ataque real, especialmente al trabajar con direcciones de memoria.

Se ha modificado la implementación original de la construcción de cadenas, desarrollando un algoritmo de búsqueda por retroceso directamente a nivel de instrucción, que no sólo verifica la coherencia de registros, sino que lleva la cuenta de los registros accedidos y purga el árbol de búsqueda si algún *gadget* altera el resultado de otro como efecto secundario. Por ejemplo, «`mov rax, rbx; pop rbp; ret`» es una operación «`mov(rax, rbx)`» que modifica `rbp` como efecto secundario. Este cambio incrementa bastante los tiempos de cómputo, por lo que hay que purgar el espacio de búsqueda.

Así, una vez obtenidos los *gadgets* del binario utilizando el Algoritmo 1 y calculadas todas las asignaciones posibles para los registros abstractos con el Algoritmo 2 y el Algoritmo 3, los *gadgets* se agrupan por combinación de registro (purgando los que no encajen en ninguna combinación o estén subsumidos por otros) en el Algoritmo 4 y finalmente se lleva a cabo una búsqueda por retroceso de cadenas en el Algoritmo 5, en el que las ramas se podan si se trata de utilizar un registro afectado por efectos secundarios. Se incluye un ejemplo de todo el flujo de trabajo de la versión ampliada de `rop3` en la Figura 3.2, desde la extracción de los *gadgets* hasta la emisión de una *ROPchain* válida.

Algoritmo 4: ConstruirPorCombinación

Entrada: *ops*: operaciones ROPLang de la cadena; *combinaciones*; $gadgets_{op}$: gadgets por paso de la cadena; *purga*: booleano
Salida: *resultado* indexado como [*combinación*][*paso*]

resultado ← lista vacía
para cada *comb* ∈ *combinaciones* **hacer**
  $porPaso \leftarrow$ lista vacía
  para cada *paso* ∈ *ops* **hacer**
    $G \leftarrow gadgets_{op}[paso]$
    $req_{dst} \leftarrow comb[paso.dst]$
    $req_{src} \leftarrow comb[paso.src]$
    $filtrado \leftarrow \{\, g \in G : (req_{dst}$ no fijado **o** $g.dst = req_{dst})$
      y (req_{src} no fijado **o** $g.src = req_{src}) \,\}$
    ordenar($filtrado$, por número de efectos secundarios)
    si *purga* **entonces**
      Purgar(*filtrado*) `// Elimina los gadgets subsumidos`
    añadir $filtrado$ a $porPaso$
  añadir $porPaso$ a *resultado*
devolver *resultado*

Algoritmo 5: ConstruirRopChain

Entrada: *ops*: pasos de la cadena; *porComb*: listas [*combinación*][*paso*]; *combinaciones*
Salida: flujo de cadenas $[g_0, g_1, \dots]$ válidas

para cada $(comb, gadgetsComb) \in (combinaciones, porComb)$ **hacer**
  $src_{ef} \leftarrow$ por cada paso, su registro fuente concreto o "ninguno"
  Función Backtrack(*i, cadena,* $afectados$)**:**
    si $i = |ops|$ **entonces**
      emitir copia de *cadena*
      devolver
    $s \leftarrow src_{ef}[i]$
    si *s es un registro* **y** *s está marcado como alterado en* $afectados$ **entonces**
      devolver
    para cada $g \in gadgetsComb[i]$ **hacer**
      marcar como alterados los registros de $g.\Delta$ en $afectados$
      añadir g al final de *cadena*
      Backtrack($i + 1$, *cadena,* $afectados$)
      quitar g de *cadena*
      desmarcar los registros de $g.\Delta$ en $afectados$
  para cada *cadena emitida por* Backtrack(0, *cadena vacía, nada alterado*) **hacer**
    emitir *cadena*

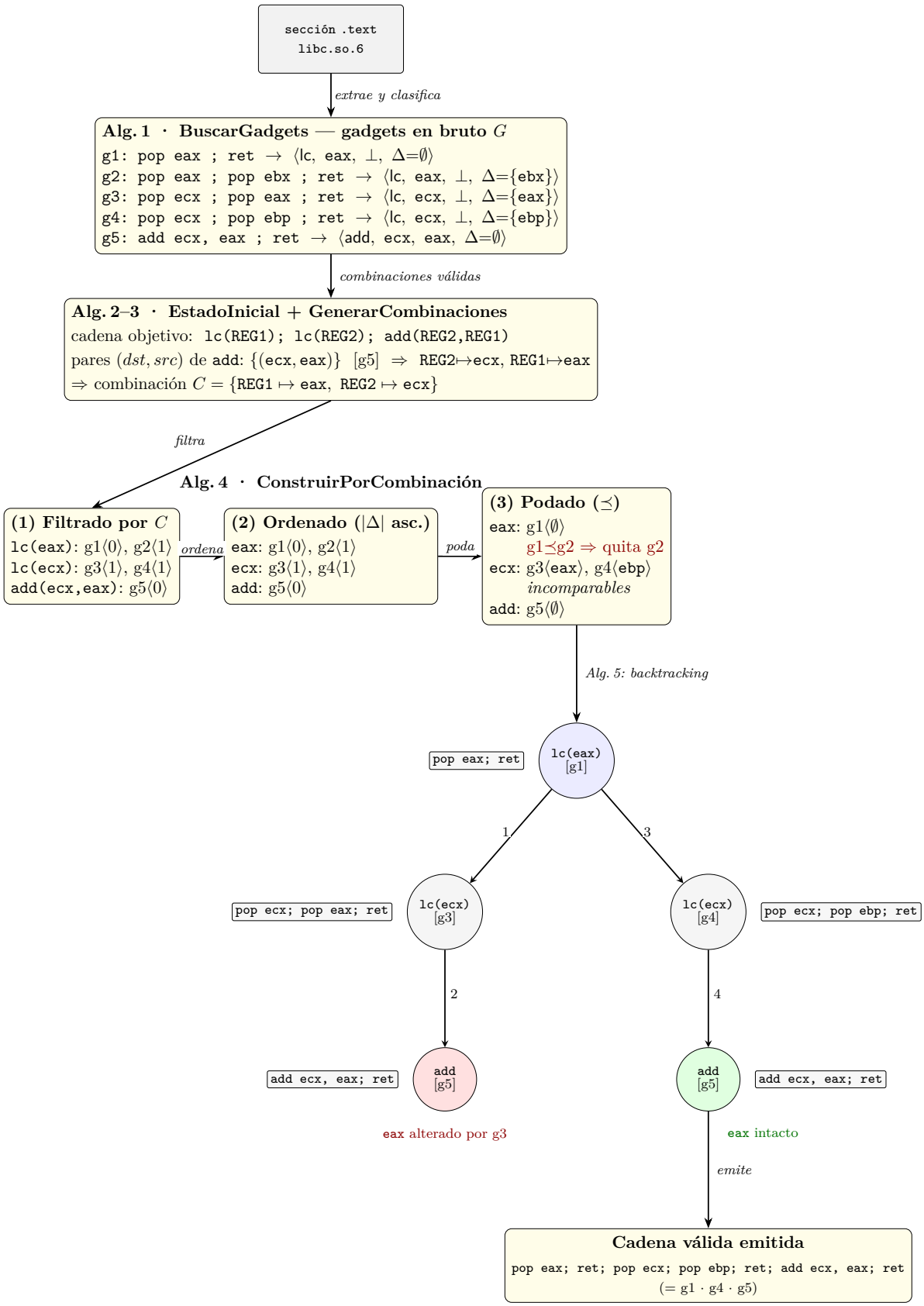

Figura 3.2: Flujo de trabajo en la búsqueda de una *ROPchain*

3.3. Completitud de Turing

La publicación original de `rop3` [2] demostraba la posibilidad de implementar una máquina de Turing utilizando las operaciones del lenguaje. No obstante, si bien esta implementación era correcta, no resultaba práctico encontrar una *ROPchain* equivalente en un binario real, ni tampoco almacenar la tabla de transiciones directamente en el *payload*.

Siguiendo el enfoque del trabajo original [2], basado a su vez en [16], se define una máquina de Turing $\mathcal{M}$ como una tupla $\langle Q, \Gamma, \sigma_0, \Sigma, q_0, F, \delta \rangle$ en la que:

- Q es un conjunto de estados finito y no vacío.
- Γ es el conjunto finito de los símbolos del alfabeto.
- σ_0 es el símbolo vacío.
- $\Sigma \subseteq \Gamma \setminus \{\sigma_0\}$ es el alfabeto de entrada.
- $q_0 \in Q$ es el estado inicial.
- $F \subseteq Q$ es el conjunto de estados finales. Cuando se alcanza uno de estos estados, la entrada se acepta.
- $\delta : (Q \setminus F) \times \Gamma \to (\Gamma \times \{L, R\} \times Q)$ es una función de transición parcialmente definida en el dominio que determina el siguiente movimiento de la máquina. L es el desplazamiento a la izquierda y R el desplazamiento a la derecha.

Asumiendo una arquitectura `x86-64`, los símbolos y estados tienen un tamaño de 8 bytes, al igual que los registros empleados. En la zona de memoria en la que se puede escribir, se reserva un espacio para la cinta y otro para la tabla de transiciones. Como ROPLang no dispone de operaciones de multiplicación, la forma más sencilla de almacenar la tabla es con una doble referencia, es decir, que una primera sección de la tabla direccione las filas y luego se localice la columna en la fila correspondiente.

Se reservan tres registros para almacenar la posición en la cinta (inicializada en el centro del vector), la dirección de la primera posición en la tabla (primera entrada de la zona de direccionamiento) y el estado actual. La condición de salida de la máquina de Turing se implementa agregando un campo adicional a cada entrada de la tabla con una dirección de salto, que será la del inicio del bucle en el caso de los estados no finales, y una dirección de salida en el caso de que se trate de un estado final. El Código 3.2 describe la cadena de instrucciones ROPLang que permite implementar la máquina descrita.

Podría objetarse que la utilidad de implementar una máquina de Turing como una *ROPchain* es mínima, salvo para probar la completitud de Turing de ROPLang. No obstante, es una medida teórica interesante que permite ilustrar una forma de ejecución condicional en ROP prescindiendo de las operaciones condicionales específicas de ROPLang.

Código 3.2: Cadena ROP para simular una máquina de Turing

```
;
;     Máquina Turing en ROP
;
lc(reg1) ; reg1 apunta a la cabeza de la cinta
lc(reg2) ; reg2 es el estado actual (escalado a 32 bytes)
lc(reg3) ; reg3 apunta al inicio de la tabla de transiciones
; --- Lectura de la cinta ---
ld(reg4, reg1)  ; reg4 = símbolo leído (escalado a 8)
add(reg4, reg3) ;
ld(reg4, reg4)  ; reg4 = dirección base de tabla[símbolo, 0]
; --- Direccionamiento por estado ---
add(reg4, reg2) ; reg4 = dirección de tabla[símbolo, estado]
; --- Ejecución del struct ---
; 1. Carga el siguiente estado
ld(reg2, reg4)
add(reg4, 8)    ; reg4 apunta al campo del struct <símbolo>
; 2. Carga el símbolo y lo escribe en la cinta
ld(reg5, reg4)
st(reg1, reg5)
add(reg4, 8)    ; reg4 apunta al campo del struct <movimiento>
; 3. Carga y aplica el movimiento de la cinta
ld(reg5, reg4)  ; +8 o -8
add(reg1, reg5)
add(reg4, 8)    ; reg4 apunta al campo del struct <@salto>
; 4. Se carga la dirección de salto (inicio del bucle o salida)
ld(reg5, reg4)
; Salto incondicional
jmp(reg5)
```

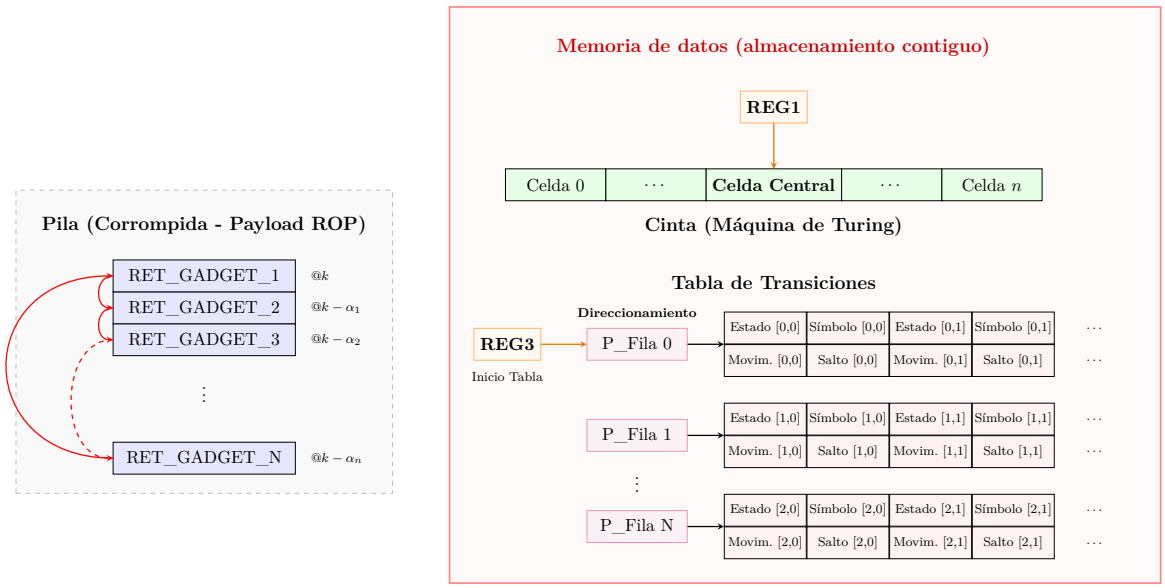

Figura 3.3: Diagrama de la máquina de Turing mediante ROP

Capítulo 4

Evaluación

En este capítulo, se describen las pruebas de capacidad de ejecución que se van a realizar en sistemas Unix. En primer lugar, se justifica la elección de las bibliotecas estándar de C como muestra. En segundo lugar, se describe el entorno de experimentación construido. En tercer lugar, se describen las *ROPchains* que se emplearán junto con los *gadgets* para medir el poder de ejecución. Finalmente, se exponen y discuten los resultados.

4.1. Reutilización de código en bibliotecas

Una de las mejores maneras de calcular la capacidad de reutilización de código de un atacante en un sistema sin considerar un programa en concreto es buscar y clasificar los *gadgets* presentes en las bibliotecas más enlazadas en el sistema, ya que constituyen una fuente estable y unificada de *gadgets*. En el contexto del software moderno, casi todos los programas que utilizan funciones importadas de bibliotecas las enlazan dinámicamente, esto es, referencian los símbolos de las funciones de la biblioteca en el ejecutable para que el enlazador pueda importarlos durante la ejecución. Por tanto, el código que se importa en la ejecución es el de la biblioteca instalada en el sistema, que de este modo se vuelve una fuente común de *gadgets* en software muy distinto.

En el caso de Windows, las bibliotecas enlazadas dinámicamente reciben el nombre de DLLs, y las más utilizadas se recogen en un conjunto llamado `KnownDLLs`. El trabajo original de `rop3` [2] utilizaba este conjunto y algunas DLLs más para medir el potencial de ejecución en este sistema operativo.

Siguiendo una aproximación similar, se escoge la biblioteca estándar de C como representante del poder de ejecución en sistemas Unix. La elección se justifica porque esta biblioteca implementa funcionalidades esenciales del sistema, lo que hace que la mayor parte del software que se ejecuta en un sistema Unix la enlace, convirtiéndola así en una fuente asegurada de ROP *gadgets*.

4.2. Entorno de experimentación

Se construye un entorno de experimentación para medir la presencia de ROP *gadgets* y el poder de ejecución en las bibliotecas estándar de C de varias distribuciones Linux, sistemas BSD y macOS. La Tabla 4.1 muestra las versiones escogidas de cada biblioteca, que corresponden al último trimestre de 2025. Los programas empleados en el entorno se implementan en Python 3 y POSIX *shell* y están disponibles en [17].

Sistema operativo	Biblioteca	Versión	Arquitecturas
Debian 13	glibc	2.41-12	`x86`, `x86-64`
Ubuntu 24.04 LTS	glibc	2.41-6ubuntu1	`x86`, `x86-64`
Fedora 43	glibc	2.42-4.fc43	`x86`, `x86-64`
Slackware 15.0	glibc	2.33	`x86`, `x86-64`
Void Linux	glibc	2.41_1	`x86`, `x86-64`
Void Linux	musl	1.2.5_1	`x86`, `x86-64`
Arch Linux	glibc	2.42+r33+gde1fe81f4714-1	`x86-64`
FreeBSD 14.3	`libc.so.7`	14.3-RELEASE	`x86`, `x86-64`
OpenBSD 7.8	`libc.so.102.0`	7.8	`x86`, `x86-64`
NetBSD 10.1	`libc.so.12`	10.1	`x86`, `x86-64`
macOS Sonoma	Varias	14.8.2	`x86-64`

Tabla 4.1: Versiones utilizadas de la biblioteca de C de cada sistema

Para las distribuciones de Linux Debian, Ubuntu, Fedora, Slackware, Void Linux y Arch Linux, se utilizan sus propios repositorios públicos, de los que es posible descargar los binarios compilados de la biblioteca estándar de C directamente, identificados por su código de versión. Se han descargado los binarios más actualizados de la versión estable de cada distribución, de haberla, o de la versión más actualizada en las distribuciones de actualización continua como Void Linux y Arch Linux. Como Arch Linux no soporta oficialmente la arquitectura `x86`, sólo se considera su versión de 64 bits de la biblioteca.

Los sistemas BSD distribuyen su software en conjuntos para el sistema base y *ports* para el código de terceros. En el caso de la biblioteca estándar de C, es necesario descargar el conjunto base del sistema, descomprimirlo y extraer la biblioteca. Para esta familia, se seleccionan los sistemas más populares: FreeBSD, OpenBSD y NetBSD.

El sistema macOS no distribuye públicamente sus binarios ni tiene una biblioteca de C monolítica, sino que la fragmenta en varias más pequeñas. Para extraerlas, se requiere disponer de un sistema macOS funcionando y de programas específicos para volcarlas de una caché interna. Específicamente, para este trabajo se extraen las bibliotecas `libsystem_c`, `libsystem_kernel`, `libsystem_malloc` y `libsystem_pthread` de 64 bits, ya que no hay versiones actualizadas de macOS en 32 bits.

Se considera que la muestra de bibliotecas escogida es suficientemente representativa de los sistemas Unix. En el caso de BSD, se incluyen los tres sistemas más utilizados, ya que cada uno incorpora su propia versión de la biblioteca estándar de C. Por otro lado, macOS es el Unix comercial más usado y su versión Sonoma todavía recibe actualizaciones de seguridad. Finalmente, la elección de distribuciones Linux resultó ser la más compleja. Escoger las más populares no es un buen criterio, ya que muchas de ellas derivan de otras distribuciones base para ajustarse a un determinado entorno gráfico o usuario, sin que esto tenga influencia real en la biblioteca de C. Por ello, se escogieron las distintas distribuciones base de las que derivan casi todas las distribuciones. Un caso particular es Void Linux, que pese a no ser muy popular resulta interesante al ofrecer dos versiones de la biblioteca estándar de C: la clásica de GNU, presente en la mayoría de las distribuciones, y `musl`, más compacta y simple.

Adicionalmente, se obtiene una muestra de bibliotecas en la distribución Debian, en su versión estable de 64 bits. Esta muestra permite buscar *ROPchains* más complejas que las que pueden localizarse en una biblioteca estándar de C, al contener binarios de mayor tamaño. Se escogen las siguientes bibliotecas, utilizando como criterio su tamaño

y popularidad en las encuestas de Debian [18]: `libcurl` (del programa curl), `libxul` (biblioteca principal de Firefox), `firefox-bin` (binario principal de Firefox), `nginx-bin` (binario principal de Nginx), `libssl` (biblioteca criptográfica), `libz` (biblioteca de compresión), `libsqlite3` (biblioteca de sqlite3), `libmergedlo` (biblioteca de LibreOffice), `libwebkit2gtk` (motor web de navegadores como GNOME Web), `libLLVM` (biblioteca principal de LLVM), `libnghttp2` (biblioteca principal de Nghttp2), `libglib-2.0` (biblioteca de propósito general utilizada por el proyecto GTK) y `libstdc++` (biblioteca estándar de C++ de GNU). Las versiones escogidas corresponden al segundo trimestre del año 2026. La Tabla 4.2 muestra las versiones concretas de cada biblioteca.

Finalmente, en este trabajo se repiten las pruebas sobre las bibliotecas de Windows de [2], con el objetivo de evaluar cómo han influido los cambios introducidos en `rop3` en el número de *gadgets* detectados. Se hace únicamente sobre las bibliotecas de la versión de 64 bits de Windows 10, ya que Windows 7 está obsoleto [19] y la versión de 32 bits de Windows 10 ha perdido mucha popularidad [20].

Paquete	**Versión**	**Biblioteca / binario**
`libcurl4t64`	8.14.1-2+deb13u3	`libcurl.so.4`
`firefox-esr`	140.11.0esr-1~deb13u1	`libxul.so`
`firefox-esr`	140.11.0esr-1~deb13u1	`firefox-bin` (binario)
`nginx`	1.26.3-3+deb13u5	`nginx` (binario)
`libssl3t64`	3.5.6-1~deb13u1	`libssl.so.3`
`zlib1g`	1.3.1-1	`libz.so.1`
`libsqlite3-0`	3.46.1-7+deb13u1	`libsqlite3.so.0`
`libreoffice-core`	25.2.3-2+deb13u3	`libmergedlo.so`
`libwebkit2gtk-4.1-0`	2.50.6-1~deb13u1	`libwebkit2gtk-4.1.so.0`
`libllvm19`	19.1.7-3	`libLLVM.so.19.1`
`libnghttp2-14`	1.64.0-1.1+deb13u1	`libnghttp2.so.14`
`libglib2.0-0t64`	2.84.4-3~deb13u2	`libglib-2.0.so.0`
`libstdc++6`	14.2.0-19	`libstdc++.so.6`

Tabla 4.2: Versiones utilizadas de las bibliotecas de Debian

4.3. Búsqueda de cadenas

Una de las funcionalidades principales de `rop3` es la definición y búsqueda de *ROPchains* o cadenas. Como se ha explicado anteriormente, una cadena es una secuencia de ROP *gadgets* con una finalidad compleja, como deshabilitar una protección o hacer una llamada al sistema. Desde el punto de vista del atacante, `rop3` es una herramienta útil a la hora de elaborar y buscar cadenas efectivas en la explotación de binarios, permitiendo incluso definir parte del ataque en abstracto, sin tener que personalizarlo para un sistema o un software específico. Con esta idea en mente, se han diseñado varias cadenas con patrones comunes de ataque y se ha intentado localizarlas en varias bibliotecas distribuidas en Debian.

El Código 4.1 realiza una llamada al sistema (*syscall* en inglés) personalizada para preparar la explotación. Las *syscalls* son llamadas a subrutinas que el núcleo del sistema

operativo expone para que los procesos no privilegiados puedan realizar acciones privilegiadas en el sistema de forma controlada. Para lograr una llamada al sistema, es necesario cargar el número de la llamada y los valores de los argumentos en una serie de registros y retornar a una instrucción ensamblador `syscall` al final de la cadena. Encadenar *syscalls* es un mecanismo común para escalar privilegios, ya que son ejecutadas directamente por el núcleo. Por tanto, modificar los argumentos de las llamadas al sistema es un requisito básico para explotar vulnerabilidades del sistema.

En el Código 4.2 se realiza una llamada personalizada a una función de biblioteca, más específicamente, a la estándar de C. Se basa igualmente en cargar valores personalizados en algunos registros y saltar finalmente a una instrucción `call` específica. Llamar arbitrariamente a las funciones de la biblioteca estándar es muy útil, ya que implementan muchas funcionalidades básicas que el atacante podría aprovechar sin tener que definirlas directamente en la cadena.

El Código 4.3 pivota el puntero de pila, una acción muy común en ROP, ya que resulta de gran utilidad cambiar el marco de pila por uno distinto en el que insertar *payloads* más grandes. Adicionalmente, se encarga también de guardar el puntero de pila anterior para permitir recuperar el estado una vez se ha terminado la explotación.

En el Código 4.4, se escribe un decodificador, esto es, un programa que almacena su propio código oculto o enmascarado y lo revela en ejecución. Esta cadena podría tener como objetivo la creación de otras *ROPchains* más difíciles de detectar, buscando con ello eludir algunas técnicas de detección basadas en vigilar direcciones de salto específicas o valores peligrosos en la pila. Se trata de una técnica más utilizada en ataques clásicos que en ROP, pero se ha escogido porque permite explotar el potencial de la búsqueda en registros abstractos de `rop3`. Adicionalmente, se prueba la cadena de la máquina de Turing del Código 3.2.

Observando las *ROPchains* escogidas, destaca la ausencia de ejecución condicional. Esto se debe a que pocos ataques ROP implementan ejecución condicional en la propia cadena. En su lugar, utilizan otros métodos, como tablas de memoria o cargar direcciones de retorno dependiendo de una lectura en memoria usando los valores condicionales como desplazamiento, de manera similar a la máquina de Turing implementada. De hecho, muchos ataques evitan cualquier forma de ejecución condicional a nivel ROP, por su complejidad y por la existencia de mecanismos más eficaces, como seleccionar previamente la cadena a ejecutar entre varias posibles.

Código 4.1: Cadena ROP para preparar los registros para una llamada al sistema

```
; Configuración de registros para una llamada al sistema en Linux
    con 3 argumentos
lc(rdi)
lc(rsi)
lc(rdx)
lc(rax) ; rax = número de la llamada
; syscall
```

Código 4.2: Cadena ROP para preparar los registros para una llamada a la biblioteca de C

```
; Configuración de registros para una llamada a libc con 4 args.
lc(rdi)
lc(rsi)
lc(rdx)
lc(rcx)
```

```
; call __libc_function__
```

Código 4.3: Cadena ROP para pivotar el puntero de pila

```
; Redirige el puntero de pila a memoria controlada por el
   atacante y luego lo restaura
gsp(reg1) ; Guarda el SP (para volver a él)
; Carga el nuevo valor del SP
lc(reg2)
jmp(reg2)
; acciones intermedias
jmp(reg1) ; Restaura el SP
```

Código 4.4: Cadena ROP para decodificar en memoria (repetición manual)

```
; decodificador
lc(reg1) ; reg1 guarda la máscara XOR
; inicio del bloque
lc(reg2)
lc(reg3)
xor(reg2, reg1)
xor(reg3, reg1)
st(reg2, reg3)
; repetir el bloque
```

4.4. Resultados

Para la obtención de resultados, se ha fijado una profundidad (tamaño máximo de bytes que puede ocupar el *gadget* sin considerar la instrucción de retorno) de búsqueda de *gadgets* de 10 y sólo se consideran terminaciones válidas para los *gadgets* las instrucciones `ret` (se descarta la instrucción `retf`, que es más problemática). Se han deshabilitado los *gadgets* que contienen saltos condicionales intermedios (`je`, `jne`, etc.) y en el caso de las operaciones ROPLang de referencia `ld` y `st`, tampoco se han considerado aquellas que utilizan dos registros para referenciar memoria (por ejemplo, `[rax + rdi*4]`). Los *gadgets* repetidos (aquellos formados por exactamente las mismas instrucciones) se cuentan una única vez. En el caso de las *ROPchains*, los registros abstractos siempre se resuelven con registros de 64 bits.

Escoger una representación adecuada para los *gadgets* en las bibliotecas no es trivial. Se considera que la representación con mapas de calor, que se utilizó en [2], resulta bastante apropiada. Permite obtener una imagen rápida de las diferencias en el número de *gadgets* entre sistemas operativos, y además, al incluir también el número exacto de *gadgets* encontrados para cada operación ROPLang y biblioteca, permite extraer conclusiones complejas y detalladas. Se decide que la normalización del color del mapa sea por operación ROPLang, ya que hacerlo así permite establecer comparaciones razonables entre sistemas operativos. En los mapas de calor, las bibliotecas estándar de C en Unix se agrupan por sistemas en el orden de la Tabla 4.1. En el caso de los resultados en Windows, se mantiene el orden por tamaño de la biblioteca del artículo original de `rop3`.

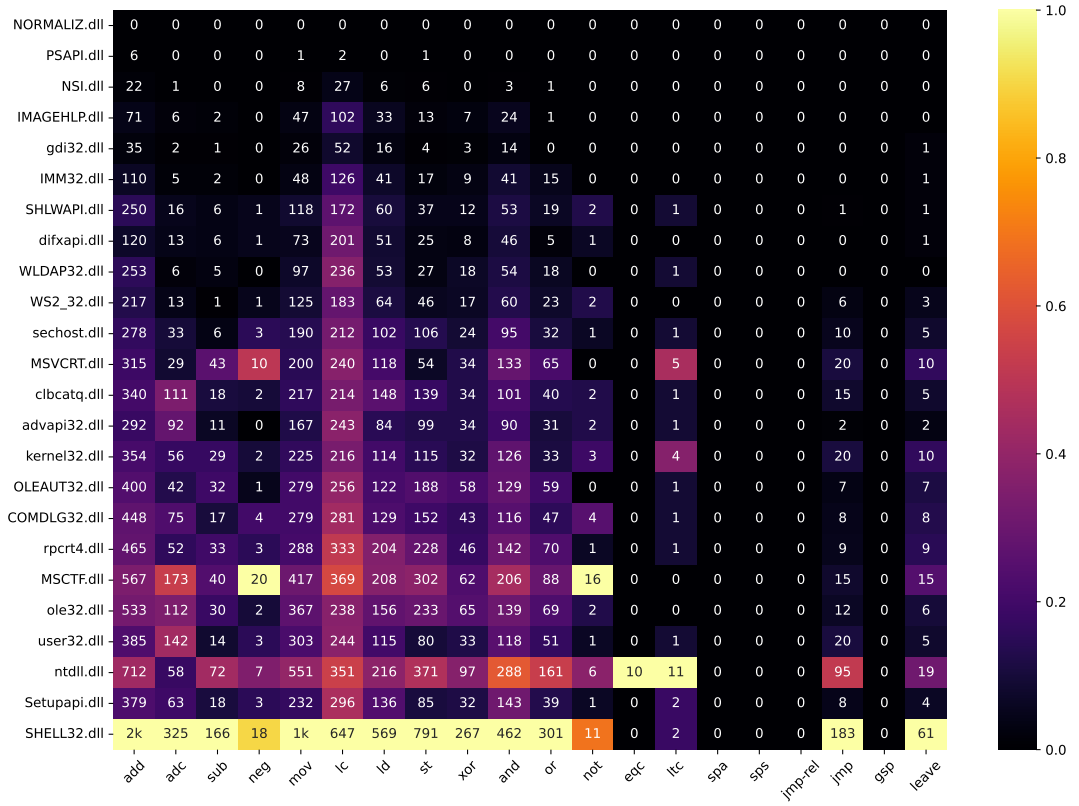

Figura 4.1: Mapa de calor de ROP *gadgets* en DLLs de Windows 10 64 bits

4.4.1. Resultados en Windows

Las muestras obtenidas en el artículo original en Windows mostraban (i) que las bibliotecas con más *gadgets* eran las de mayor tamaño, (ii) un número significativo de operaciones aritméticas y lógicas y (iii) un número reducido de operaciones condicionales y de salto.

La Figura 4.1 revela que los resultados en Windows son consistentes con el trabajo original, ya que muestra que sólo se ha producido una mínima reducción global del número de *gadgets* debido a que en la nueva versión de `rop3` se consideran inválidos algunos *gadgets* que antes no lo eran. Las operaciones **`spa`**, **`sps`** y **`jmp-rel`** han desaparecido por completo, ya que ahora sólo se cuentan como tales las que utilizan como registro fuente un registro del procesador, descartando las que utilizan valores inmediatos.

4.4.2. Resultados en Unix

La Figura 4.2 revela una amplia presencia de *gadgets* esenciales en casi todas las bibliotecas, tanto de Linux como de BSD (salvo OpenBSD) y macOS. Dado que en estas pruebas los registros abstractos en *ROPchains* se resuelven siempre con registros del máximo tamaño de la arquitectura, todas las operaciones compuestas de ROPLang buscan que los registros abstractos empleados sean de 64 bits. Esto explica la llamativa inexistencia de *gadgets* en operaciones como **`spa`** o **`sps`**, cuyo número es ahora verdaderamente representativo de la acción que se desea realizar (por ejemplo, el objetivo de **`spa`** era lograr ejecución condicional y, por tanto, sumar valores indirectos al puntero de pila). Se encuentran abundantes *gadgets* de operaciones aritméticas, lógicas y de memoria.

Un caso muy distinto son las operaciones de ejecución condicional **`eqc`** y **`ltc`**, que están

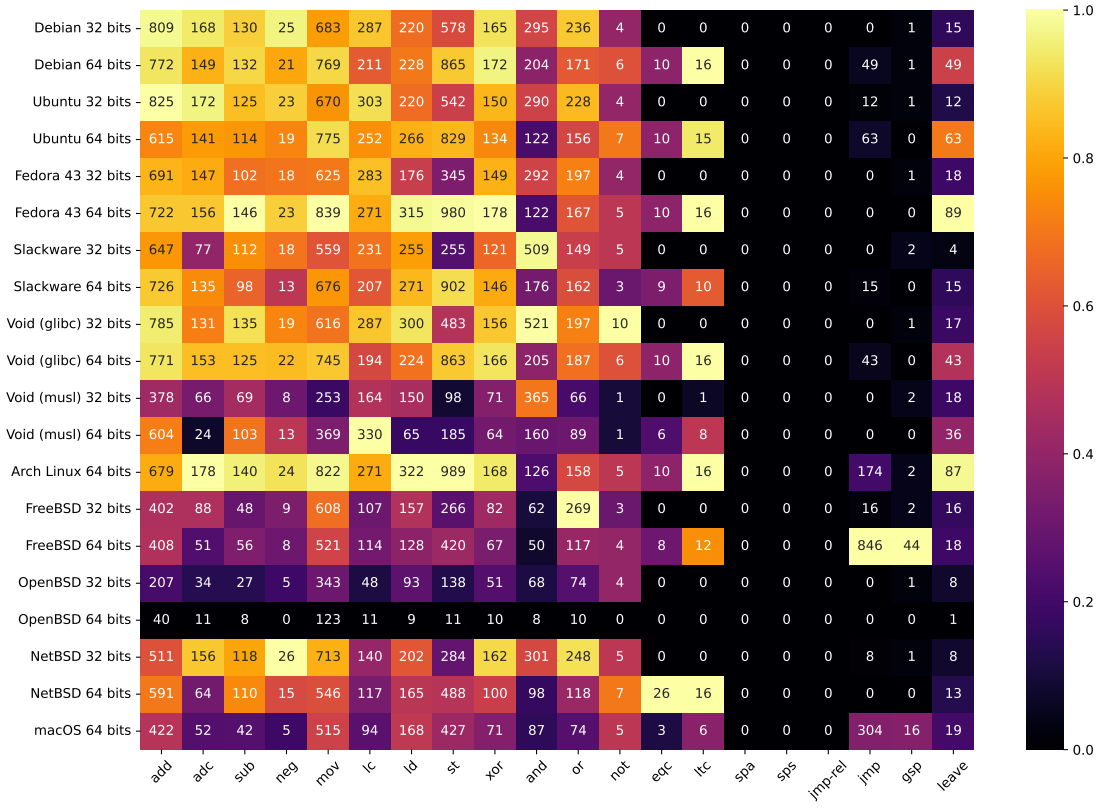

Figura 4.2: Mapa de calor de ROP *gadgets* en bibliotecas estándar de C

distribuidas desigualmente y son inexistentes en algunas bibliotecas. La operación de salto **`spa`** no se encuentra en ninguna biblioteca. Dado que **`jmp-rel`** (la antigua operación de salto) es un alias de **`spa`**, se puede decir lo mismo de ella. La operación **`jmp`** de salto absoluto es relativamente escasa y no está presente en muchas bibliotecas. La operación **`gsp`** es todavía más escasa que **`jmp`**. Esta es una operación relevante en muchos ataques que implican intercambiar o guardar el puntero de pila, por lo que su presencia es importante, a pesar de no ser estrictamente necesaria para garantizar completitud de Turing de ROPLang. Los resultados en estas operaciones parecen confirmar la superioridad del nuevo método de salto y ejecución condicional de la Subsección 3.2.2.

Finalmente, **`leave`** se encuentra en todas las bibliotecas. Dado que no admite múltiples versiones (opera siempre con los mismos registros), la existencia de unos pocos *gadgets* es esperable al eliminar duplicidades, por lo que los *gadgets* encontrados son más que suficientes.

Una vez analizados los *gadgets* globalmente, es posible sacar conclusiones sobre las diferencias entre sistemas operativos. Lo primero que llama la atención es que los resultados en sistemas Linux con la biblioteca estándar de C de GNU `glibc` son muy similares, incluso al comparar sus versiones de 32 y 64 bits, dado que únicamente presentan diferencias significativas en algunas operaciones condicionales. Al tratarse de versiones similares compiladas generalmente con el compilador GCC, no resulta demasiado sorprendente este parecido. En el caso de Void Linux, la diferencia entre la implementación `glibc` y `musl` no es tan pronunciada y se puede explicar por el menor tamaño de esta última.

Como cada uno de los sistemas BSD y macOS implementa la biblioteca estándar de C de forma distinta, es normal que no haya tanto parecido con las versiones de Linux ni entre ellas. No obstante, dado que hay patrones comunes en la familia BSD, las bibliotecas de

FreeBSD y NetBSD tienen en general números similares de *gadgets*, salvo en la instrucción `jmp`, donde FreeBSD presenta un valor anómalo.

Finalmente, observando la Figura 4.2, destaca la escasa frecuencia de ROP *gadgets* en OpenBSD, sobre todo en la biblioteca de 64 bits. Esto se debe a que OpenBSD implementa a nivel de software protecciones contra ROP, utilizando una especie de guardas similares a los canarios de pila [21] (valores centinela colocados antes de la dirección de retorno para detectar modificaciones no autorizadas de la pila). A esta mitigación de seguridad se la conoce como *retguard*. Cada programa del sistema base que se ejecuta en OpenBSD dispone de una sección de múltiples *retguards* teóricamente no visibles para un potencial atacante, con los que se codifica mediante un `xor` la dirección de retorno al entrar en la función. Justo antes de ejecutar cualquier instrucción `ret`, se invierte la operación y se verifica que la dirección de retorno no ha sido modificada. En principio, esta técnica impide la realización de ataques ROP, ya que cualquier potencial *gadget* (excepto los no alineados) interrumpiría la ejecución justo antes de llegar a la instrucción `ret`.

No obstante, el mecanismo descrito no explica por sí solo los resultados obtenidos. Tras examinar el código fuente de la biblioteca de C, se ha observado que OpenBSD introduce un *padding* de instrucciones de interrupción entre la verificación del canario y la instrucción `ret`. Por tanto, para encontrar más *gadgets* sería necesario aumentar mucho la profundidad de búsqueda o preprocesar el binario para eliminar estos *paddings* e incluso *retguard* en su conjunto.

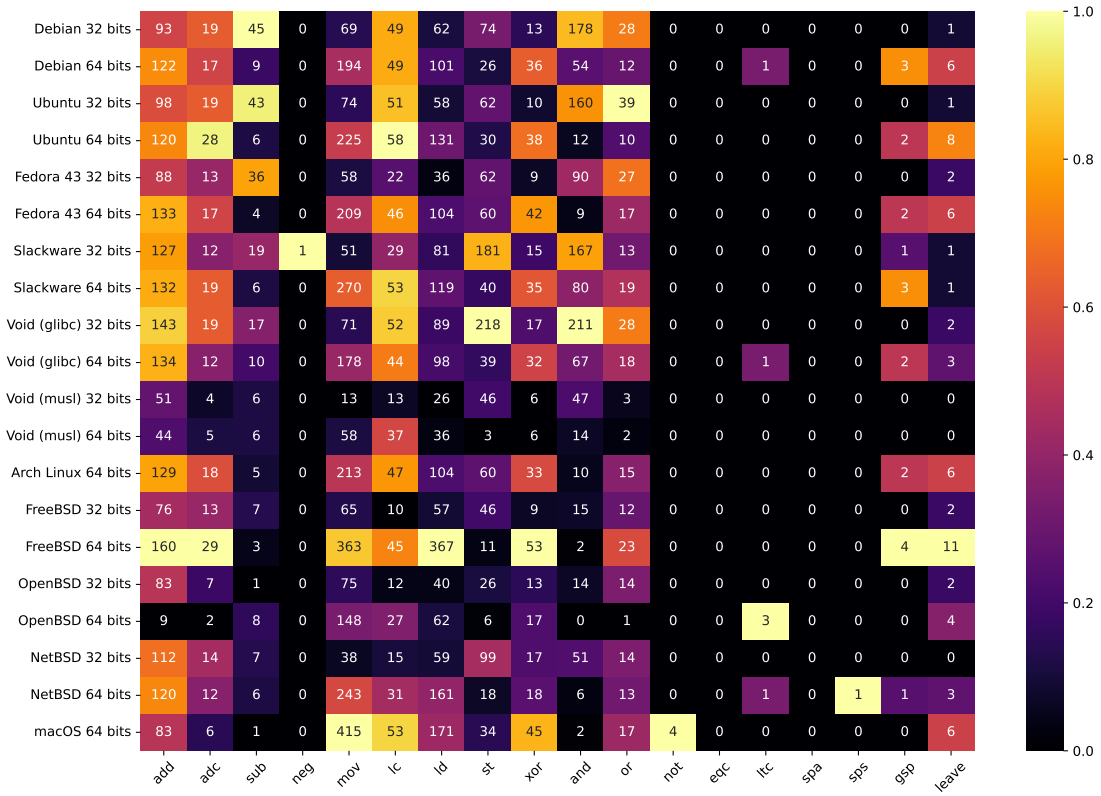

Figura 4.3: Mapa de calor de JOP *gadgets* en bibliotecas estándar de C

Programación orientada al salto. La programación orientada al salto (JOP, del inglés *Jump Oriented Programming*) es una técnica similar a ROP útil en entornos donde la manipulación de la dirección de retorno en la pila no resulta posible. Al igual que ROP,

encadena pequeños fragmentos de código que finalizan en un salto (`jmp` o `call`) con un operando indirecto, como puede ser un registro o una dirección en memoria. Adicionalmente, suele requerir de un *gadget* central denominado *dispatcher* encargado de encadenar la secuencia de *gadgets*, emulando el comportamiento de un puntero de instrucción.

Dado que el mecanismo de localización de JOP *gadgets* en `x86`/`x86-64` es esencialmente el mismo que para encontrar ROP *gadgets* (buscar los bytes correspondientes a una instrucción de control de flujo, como `ret` o `jmp`, y leer las instrucciones anteriores), `rop3` se extiende fácilmente a esta técnica de reutilización de código, lo que permite extraer mapas de calor de JOP *gadgets* igual que en el caso de ROP. La Figura 4.3 muestra la cantidad de JOP *gadgets* localizados en cada biblioteca.

Los resultados muestran que hay globalmente un número menor de JOP *gadgets* que de ROP *gadgets*, lo que, sumado al hecho de que no es trivial combinar *gadgets* entre sí, hace que las posibilidades de realizar un ataque JOP en las bibliotecas de los sistemas Unix analizadas sean realmente limitadas. Parece haber un número suficiente de operaciones aritméticas, lógicas y de acceso a memoria en la mayoría de las bibliotecas, pero las operaciones relacionadas con la bandera de acarreo y con el puntero de pila son escasas.

Un *gadget* de gran importancia en JOP es el que suma un valor a un registro, ya que es uno de los principales mecanismos para encadenar JOP *gadgets*. En ROPLang, este *gadget* es `add`, que parece estar presente en todas las bibliotecas.

En términos generales, puede concluirse que la explotación mediante JOP en las bibliotecas estándar de C es mucho más compleja que su equivalente ROP, aunque no imposible. Además, la escasez de operaciones que operan con el puntero de pila, como `gsp` o `spa`, dificulta la transición de JOP a ROP, un procedimiento relativamente común.

4.4.3. Resultados de la búsqueda de *ROPchains*

Para finalizar, los resultados de la Tabla 4.3 muestran qué bibliotecas presentan una secuencia de operaciones capaz de ejecutar correctamente cada *ROPchain* definida en la Sección 4.3. Estos resultados evidencian la presencia de patrones comunes de ataque ROP en muchas bibliotecas y binarios de Debian. La identificación exitosa de la *ROPchain* de decodificación en varias bibliotecas revela también el potencial de abstracción de `rop3`

Binarios \ ROPchains	Tamaño (MB)	decoder	ret2libc	stack_pivoting	syscall	turing
`libz.so.1`	0,12	✗	✗	✗	✗	✗
`libnghttp2.so.14`	0,19	✗	✗	✗	✗	✗
`firefox-bin`	0,55	✗	✓	✗	✓	✗
`libcurl.so.4`	0,94	✗	✓	✗	✓	✗
`libssl.so.3`	1,05	✗	✗	✗	✗	✗
`libglib-2.0.so.0`	1,34	✗	✓	✗	✓	✗
`nginx`	1,42	✗	✓	✗	✓	✗
`libsqlite3.so.0`	1,5	✗	✓	✗	✓	✗
`libstdc++.so.6`	2,38	✗	✓	✗	✓	✗
`libmergedlo.so`	82,53	✗	✓	✓	✓	✓
`libwebkit2gtk.so.0`	87,47	✓	✓	✓	✓	✓
`libLLVM.so.19.1`	123,67	✓	✓	✓	✓	✓
`libxul.so`	156,5	✓	✓	✓	✓	✓

Tabla 4.3: Cadenas encontradas en bibliotecas de Debian (ordenadas por tamaño)

a la hora de elaborar cadenas complejas. Haber localizado potenciales máquinas de Turing en las cuatro bibliotecas de mayor tamaño es un resultado de gran importancia para medir el poder de ejecución, ya que extiende el resultado teórico de la completitud de Turing de ROPLang a un resultado experimental sobre la existencia de *gadgets* reales capaces de ejecutar una máquina de Turing en bibliotecas grandes. Por último, se verifica la hipótesis de [1, 9] de que los binarios de gran tamaño son susceptibles de albergar conjuntos Turing-completos de *gadgets*.

Capítulo 5

Estado del arte

En los últimos años, la carrera armamentística en las técnicas ROP y JOP ha continuado, en un escenario de crecientes mitigaciones a nivel hardware que han complementado y en algunos casos superado a las mitigaciones software. En este capítulo, se expone el estado del arte en ROP, tanto desde el punto de vista de los atacantes como de los defensores, y se señalan las aportaciones de `rop3`.

5.1. Atacantes

Las protecciones clásicas contra ROP, como ASLR [11] (del inglés *Address Space Layout Randomization*, destinada a aleatorizar la disposición en memoria de las secciones del binario) o la integridad del flujo de control [22] (destinada a asegurar que todas las transiciones que se producen en el programa se corresponden con la estructura legítima del grafo de flujo de control) han dificultado mucho los ataques de reutilización de código, pero no han sido capaces de solucionarlos por completo. El desarrollo de técnicas ofensivas basadas en la programación orientada al retorno ha seguido cuatro direcciones claramente definidas: (i) superar la aleatorización del espacio de memoria y las medidas de integridad del flujo de control, (ii) lograr un mayor grado de abstracción en la construcción de las cadenas, (iii) demostrar el potencial ofensivo de ROP para realizar acciones complejas y (iv) extender la metodología ROP a otras formas de reutilización de código.

En la primera dirección, se ha tratado de superar defensas software como ASLR en Unix. Dado que esta mitigación se limita a aleatorizar la disposición en memoria del código, pero no a alterarlo como tal, los atacantes han buscado formas de leer la memoria del programa durante su ejecución para localizar *gadgets* en tiempo real. Por ejemplo, el trabajo de [3] revisita diez años después el artículo original sobre ROP [1] para demostrar el potencial del análisis dinámico de binarios para superar las defensas software que se habían desarrollado hasta el momento, como ASLR o técnicas de realeatorización.

En la segunda dirección, se han publicado artículos como [23], que utilizan la representación intermedia de programas de ingeniería inversa para localizar ROP *gadgets*. Otras herramientas [24, 25] también han seguido enfoques similares.

En el contexto concreto del sistema operativo Windows, en [26] se ha demostrado el gran potencial de ROP para diseñar ataques complejos usando la API de Windows y ROP puro, en lugar de utilizarlo únicamente para deshabilitar DEP, como es habitual en este sistema. En este trabajo, se identifican y clasifican patrones de *gadgets* relacionados con funcionalidades específicas para la explotación de la API de Windows. La herramienta `rop3` también permite en gran medida calcular el potencial de ejecución de los binarios de

un sistema, pero carece de la especificidad de [26].

Finalmente, tras la publicación de [1, 27], han surgido nuevas metodologías de reutilización de código. En relación con JOP, *Counterfeit Object-Oriented Programming* [28] es un método de ataque que utiliza las funciones virtuales de C++ mediante objetos manipulados para lograr ataques muy complejos y Turing-completos. En el caso de ROP en sistemas Unix, se ha desarrollado una técnica potente que permite realizar ataques complejos con pocos *gadgets*, gracias a la llamada al sistema `sigreturn` [29]. La metodología *block oriented programming* [30], por su parte, busca lograr objetivos similares a ROP, pero asumiendo las restricciones de integridad del flujo de control y trabajando directamente con bloques en lugar de *gadgets*, con el consiguiente aumento en la complejidad, pero respetando el flujo de control del programa.

Algunos trabajos [31, 32, 33] han señalado el potencial de ROP en la ofuscación de programas, ya que la mayoría de los *frameworks* de ingeniería inversa actuales son incapaces de reconstruir el flujo de control de un programa implementado mediante *gadgets* y manipulación anómala de las direcciones de retorno en la pila. Se han propuesto tanto programas de ofuscación como desofuscadores específicos contra estas técnicas [34].

5.2. Defensores

En las arquitecturas `x86`/`x86-64`, los fabricantes han introducido protecciones a nivel hardware, como Intel CET [35], orientada a garantizar la integridad del flujo de control. En el caso concreto de ROP y esta mitigación, la introducción de pilas de sombra en las que se guardan las direcciones de retorno de las funciones y donde el atacante no puede escribir mediante una manipulación normal de la pila imposibilita la mayor parte de los ataques ROP, por lo que las técnicas ofensivas han ido migrando hacia otras formas de reutilización de código. Adicionalmente, CET incluye también una protección contra JOP: los *landing pads*, que limitan las secciones de código a las que se puede saltar. Para que estas protecciones estén habilitadas, es necesario que el binario se haya compilado específicamente con soporte para ellas. Esto último todavía no es tan habitual en la familia Unix, especialmente en el caso de los BSD y en macOS.

Por su parte, la arquitectura ARM introdujo mitigaciones todavía más potentes en sus versiones más recientes. Concretamente, la autenticación de punteros [36], que reserva una parte de los bits no utilizados en un tipo puntero de 64 bits para guardar una firma, permite verificar criptográficamente las direcciones de retorno con escaso coste computacional. En la práctica, esto dificulta enormemente la labor del atacante, que no puede introducir directamente la dirección de retorno en la pila, sino que debe buscar alguna manera de firmarla, normalmente explotando otra vulnerabilidad en el programa o en el diseño de la firma de punteros [37], algo que en general resulta muy complicado. Esta arquitectura incluye también su propia implementación de *landing pads* mediante la instrucción `bti` [38].

El sistema macOS ha sabido aprovechar muy bien el cambio de arquitectura, ya que implementa autenticación de punteros y *landing pads* en su código más crítico [39, 40]. Lo anterior, unido al hecho de que la arquitectura `arm64` es alineada (a diferencia de `x86`/`x86-64`), hace improbable un ataque ROP en software crítico en un macOS moderno. A pesar de todo, la autenticación de punteros y los *landing pads* no están habilitados en todo el código que ejecuta el sistema operativo, ya que la mayoría de las aplicaciones de terceros no se compilan con soporte para estas mitigaciones. Por tanto, la ejecución de ataques ROP y JOP sigue siendo posible, aunque restringida.

Por otro lado, muchas distribuciones Linux y sistemas BSD han incluido también

soporte para verificar las direcciones de retorno y los *landing pads* en arquitecturas `arm64`, pero está menos integrado que en macOS.

5.3. Aportaciones

Las innovaciones introducidas en `rop3` hacen varias aportaciones al estado del arte en ROP, tanto a nivel ofensivo como defensivo.

A nivel ofensivo, `rop3` logra un gran grado de abstracción en la búsqueda de *gadgets* mediante la introducción y perfeccionamiento de ROPLang. Como se ha visto, `rop3` no es la única herramienta capaz de lograr esta generalización de los *gadgets*, pero ofrece una serie de ventajas. Por un lado, algunos programas clásicos de búsqueda de *gadgets* como `ROPgadget` [41] o `Ropper` [42] ya implementaban técnicas similares, pero sólo eran capaces de sintetizar unas pocas *ROPchains* con funcionalidades concretas, como ejecutar una consola de comandos o deshabilitar DEP, mientras que `rop3` es capaz de sintetizar cualquier tipo de cadena programable en ROPLang. Otras herramientas publicadas, como `angrop` [24] o `SGC` [25], usan aproximaciones muy distintas basadas en el análisis simbólico y en lenguajes de representación intermedia, logrando en algunos casos mejor abstracción, pero con un elevado aumento en complejidad y tiempo, oscureciendo además parte de la semántica real de los *gadgets* y dificultando su clasificación.

A nivel defensivo, `rop3` es una herramienta claramente enfocada en revelar el potencial de ejecución de los atacantes usando ROP, ya que permite cuantificar y clasificar de forma sencilla las acciones que un atacante puede realizar mediante reutilización de código, como se ha evaluado en el Capítulo 4.

La cuantificación del potencial de ataque ROP también ha sido abordada por otros trabajos, que utilizan aproximaciones distintas basadas en métricas de calidad y caracterización de los conjuntos de gadgets [43, 44]. En términos generales, estos trabajos coinciden en buscar formas de medir la capacidad de ejecución más elaboradas que el conteo de *gadgets* agrupados por funcionalidad. A diferencia de estas aproximaciones, `rop3` prioriza un método de análisis directo y claro: al tratar cada *gadget* como una operación con una funcionalidad definida que está presente o no en el binario, ofrece una lectura inmediata e interpretable del potencial de ejecución del atacante. Además, la posibilidad de buscar *ROPchains* abstractas con funcionalidades avanzadas en binarios supone una métrica adicional de la capacidad de ejecución que se encuentra implementada en `rop3`.

Capítulo 6

Conclusiones y trabajo futuro

Este trabajo extiende el soporte de la herramienta `rop3` tanto a los formatos ejecutables de Unix como a la detección de efectos secundarios en la construcción de *ROPchains*. Adicionalmente, se han arreglado algunos problemas relacionados con la construcción de ROP *gadgets* inválidos por incluir saltos incondicionales intermedios o utilizar registros de bits reducidos para resolver los registros abstractos de la *ROPchain*.

La interpretación de resultados amplía las conclusiones del artículo original de `rop3`, demostrando la existencia de una serie de patrones que parecen ser comunes al código de bibliotecas `x86/x86-64` en general, y no específicos de un sistema operativo. Además, los cambios introducidos en la máquina de Turing han logrado localizar *ROPchains* capaces de implementar de forma práctica este instrumento clásico para determinar la capacidad de ejecución y probar experimentalmente la completitud de Turing en binarios grandes.

Todo el trabajo realizado se ha publicado bajo la licencia de software libre GPLv3 y es reproducible, permitiendo a la comunidad servirse de él y contribuyendo al desarrollo transparente de herramientas de ciberseguridad.

Respecto al trabajo futuro, el esfuerzo realizado en la formalización de los *gadgets*, de ROPLang y de los algoritmos del programa revela que gran parte de la funcionalidad de `rop3` no está anclada a ROP o JOP como método de explotación, por lo que la extensión de la herramienta a otros paradigmas de reutilización de código es un buen camino a seguir. Del mismo modo, el siguiente paso sería la extensión a otras arquitecturas hardware, como ARM o RISC-V.

La detección de efectos secundarios en memoria sigue sin estar implementada. Para resolver este problema, se ha explorado con cierto éxito una técnica basada en aprendizaje con pocas muestras [45] utilizando modelos extensos de lenguaje. Mezclar algoritmos tradicionales con aproximaciones basadas en modelos extensos de lenguaje es un proceso que ha resultado eficaz en muchos problemas que no disponían de una solución eficiente, por lo que la expansión de `rop3` por esta vía puede resultar interesante.

Bibliografía

[1] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". En: *Proceedings of the 14th ACM conference on Computer and communications security*. 2007, págs. 552-561.

[2] Daniel Uroz y Ricardo J. Rodríguez. "Evaluation of the Executional Power in Windows using Return Oriented Programming". En: *2021 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2021, págs. 361-372.

[3] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos y Cristiano Giuffrida. "The dynamics of innocent flesh on the bone: Code reuse ten years later". En: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, págs. 1675-1689.

[4] Aleph One. "Smashing the Stack for Fun and Profit". En: *Phrack* 7.49 (1996).

[5] Pavel Yosifovich, Mark E Russinovich, Alex Ionescu y David A Solomon. *Windows Internals: System architecture, processes, threads, memory management, and more, Part 1*. 7th. Microsoft Press, 2017. ISBN: 978-0-7356-8418-8.

[6] Theo de Raadt. "Exploit Mitigation Techniques". OpenCON 2005 (Venice, Italy). Presentation slides. Nov. de 2005. URL: https://www.openbsd.org/papers/ven05-deraadt/index.html.

[7] Solar Designer. *Getting around non-executable stack (and fix)*. Bugtraq mailing list. 1997. URL: https://seclists.org/bugtraq/1997/Aug/63.

[8] Nergal. "The advanced return-into-lib(c) exploits: PaX case study". En: *Phrack Magazine*. Vol. 11. 58. 2001.

[9] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler y Michael Franz. "Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming." En: *WOOT* 12 (2012), págs. 64-76.

[10] pakt. *ropc: A ROP compiler*. 25 de jun. de 2013. URL: https://github.com/pakt/ropc (visitado 01-06-2026).

[11] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu y Dan Boneh. "On the effectiveness of address-space randomization". En: *Proceedings of the 11th ACM Conference on Computer and Communications Security*. Association for Computing Machinery, 2004, págs. 298-307.

[12] Daniel Uroz, Jaime Alconchel y Ricardo J. Rodríguez. *rop3: A tool to search for gadgets, operations, and ROP chains using a backtracking algorithm in a tree-like structure*. Ver. 2.0.0. 2026. URL: https://github.com/reverseame/rop3 (visitado 22-06-2026).

[13] The Capstone Team. *Capstone Disassembly Framework*. Ver. 5.0.9. 28 de mayo de 2026. URL: https://www.capstone-engine.org.

[14] Ronald Oussoren. *macholib: Mach-O header analysis and editing*. Ver. 1.16.4. 22 de nov. de 2025. URL: https://pypi.org/project/macholib/.

[15] Eli Bendersky. *pyelftools: Pure-Python library for parsing and analyzing ELF files and DWARF debugging information*. Ver. 0.33. 29 de mayo de 2026. URL: https://pypi.org/project/pyelftools/.

[16] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena y Zhenkai Liang. "Data-oriented programming: On the expressiveness of non-control data attacks". En: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, págs. 969-986.

[17] Jaime Alconchel. *Scripts TFG ROP 2026*. Jun. de 2026. URL: https://github.com/845977/tfg_rop_2026_scripts.

[18] *Debian Popularity Contest*. Debian Project. URL: https://popcon.debian.org/ (visitado 10-06-2026).

[19] Microsoft. *Windows 7 - Microsoft Lifecycle*. URL: https://learn.microsoft.com/en-us/lifecycle/products/windows-7 (visitado 25-06-2026).

[20] PassMark Software. *Windows OS Marketshare - Usage Statistics*. URL: https://www.pcbenchmarks.net/os-marketshare.html (visitado 25-06-2026).

[21] Todd Mortimer. "Removing ROP gadgets from OpenBSD". En: *Proc. of the AsiaBSDCon* (2019), págs. 13-21.

[22] Martín Abadi, Mihai Budiu, Ulfar Erlingsson y Jay Ligatti. "Control-flow integrity principles, implementations, and applications". En: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), págs. 1-40.

[23] Mark DenHoed y Tom Melham. "Synthesis of Code-Reuse Attacks from p-code Programs". En: *34th USENIX Security Symposium (USENIX Security 25)*. 2025, págs. 395-411.

[24] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel y Giovanni Vigna. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis". En: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, págs. 138-157.

[25] Moritz Schloegel, Tim Blazytko, Julius Basler, Fabian Hemmer y Thorsten Holz. "Towards automating code-reuse attacks using synthesized gadget chains". En: *European Symposium on Research in Computer Security*. Springer. 2021, págs. 218-239.

[26] Bramwell Brizendine, Shiva Shashank Kusuma y Bhaskar P. Rimal. "Process Injection Using Return-Oriented Programming". En: *IEEE Access* 13 (2025), págs. 133790-133816.

[27] Tyler Bletsch, Xuxian Jiang, Vince W Freeh y Zhenkai Liang. "Jump-oriented programming: a new class of code-reuse attack". En: *Proceedings of the 6th ACM symposium on information, computer and communications security*. 2011, págs. 30-40.

[28] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi y Thorsten Holz. "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications". En: *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2015, págs. 745-762.

[29] Erik Bosman y Herbert Bos. "Framing Signals – A Return to Portable Shellcode". En: *2014 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2014, págs. 243-258.

[30] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger y Mathias Payer. "Block Oriented Programming: Automating Data-Only Attacks". En: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2018, págs. 1868-1882.

[31] Dongliang Mu, Jia Guo, Wenbiao Ding, Zhilong Wang, Bing Mao y Lei Shi. "ROPOB: obfuscating binary code via return oriented programming". En: *International Conference on Security and Privacy in Communication Systems*. Springer. 2017, págs. 721-737.

[32] Pietro Borrello, Emilio Coppa y Daniele Cono D'Elia. "Hiding in the particles: When return-oriented programming meets program obfuscation". En: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2021, págs. 555-568.

[33] Christoforos Ntantogian, Georgios Poulios, Georgios Karopoulos y Christos Xenakis. "Transforming malicious code to ROP gadgets for antivirus evasion". En: *IET Information Security* 13.6 (2019), págs. 570-578.

[34] Haoran Liu, Tieming Liu, Rui Chang, Jian Lin, Zuozheng Zhou y Jing Jing. "ROParser: A High-Efficient Framework for Return-Oriented Programming Deobfuscation". En: *Computers & Security* (2026), pág. 104884.

[35] Tom Garrison. "Intel CET answers call to protect against common malware threats". En: *Intel, Mountain View, CA, USA, Jun* 15 (2020).

[36] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg y N Asokan. "PAC it up: Towards pointer integrity using ARM pointer authentication". En: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, págs. 177-194.

[37] Xiaolong Bai y Min Zheng. "HackPac: Hacking Pointer Authentication in iOS User Space". En: *Proceedings of DEF CON 27*. Presented on August 9, 2019. Las Vegas, NV, ago. de 2019. URL: https://infocondb.org/con/def-con/def-con-27/hackpac-hacking-pointer-authentication-in-ios-user-space.

[38] Arm Limited. *Arm Architecture Reference Manual Supplement, The A64 Instruction Set Architecture*. DDI0602. Arm Limited. Mar. de 2026. Cap. BTI – Branch Target Identification. URL: https://developer.arm.com/documentation/ddi0602/2026-03/Base-Instructions/BTI--Branch-target-identification- (visitado 12-06-2026).

[39] Apple Inc. *Operating System Integrity*. En: *Apple Platform Security*. Apple Inc., mar. de 2026. URL: https://support.apple.com/guide/security/operating-system-integrity-sec8b776536b/web (visitado 12-06-2026).

[40] networkextension. *ARMv8.3 Pointer Authentication in XNU*. GitHub Gist. Feb. de 2021. URL: https://gist.github.com/networkextension/24bac6ffb8cb875e1d8b4f8672cdba3b (visitado 12-06-2026).

[41] Jonathan Salwan. *ROPgadget*. Ver. 7.7. 15 de oct. de 2025. URL: https://github.com/JonathanSalwan/ROPgadget (visitado 16-06-2026).

[42] Sascha Schirra. *Ropper*. Ver. 1.13.13. 28 de feb. de 2025. URL: https://github.com/sashs/Ropper (visitado 23-06-2026).

[43] Andreas Follner, Alexandre Bartel y Eric Bodden. "Analyzing the Gadgets: Towards a Metric to Measure Gadget Quality". En: *Engineering Secure Software and Systems (ESSoS)*. Vol. 9639. Lecture Notes in Computer Science. Springer, 2016, págs. 155-172.

[44] Michael D. Brown y Santosh Pande. "Is Less Really More? Towards Better Metrics for Measuring Security Improvements Realized Through Software Debloating". En: *Proceedings of the 12th USENIX Workshop on Cyber Security Experimentation and Test (CSET)*. USENIX Association, 2019.

[45] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever y Dario Amodei. "Language Models are Few-Shot Learners". En: *Advances in Neural Information Processing Systems*. Ed. por H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan y H. Lin. Vol. 33. Curran Associates, Inc., 2020, págs. 1877-1901.

Anexo A

Planificación y distribución temporal

En este anexo se describe la organización temporal del trabajo, el esfuerzo dedicado a cada una de las actividades y la evolución del proyecto a lo largo de su desarrollo. La información que aquí se presenta procede del registro de dedicación semanal que se ha mantenido desde la reunión inicial con el tutor hasta la entrega de la memoria.

A.1. Visión general del esfuerzo

El proyecto se ha desarrollado a lo largo de aproximadamente nueve meses, entre finales de septiembre de 2025 y mediados de junio de 2026. La dedicación total es de unas 311 horas de trabajo efectivo.

Categoría	**Horas**	**Porcentaje**
Memoria	95,0	30,5 %
Programación	62,0	19,9 %
Estudio	75,5	24,2 %
Técnico	50,0	16,0 %
Análisis	26,0	8,3 %
Reuniones con el tutor	3,3	1,1 %
Total	**311,8**	**100,0 %**

Tabla A.1: Distribución del esfuerzo por categoría de tarea.

A.2. Línea temporal del proyecto

Para representar la evolución del trabajo se han identificado cinco fases principales, que se solapan parcialmente en el tiempo. La Figura A.1 muestra el diagrama de Gantt correspondiente, donde cada barra abarca el intervalo en el que la actividad estuvo activa. Los rombos señalan los hitos más relevantes del proyecto.

A continuación se describe brevemente cada una de las fases:

Documentación y estudio. Comprende la lectura de los artículos de referencia sobre ROP y el estudio de las herramientas, arquitecturas binarias y mitigaciones existentes.

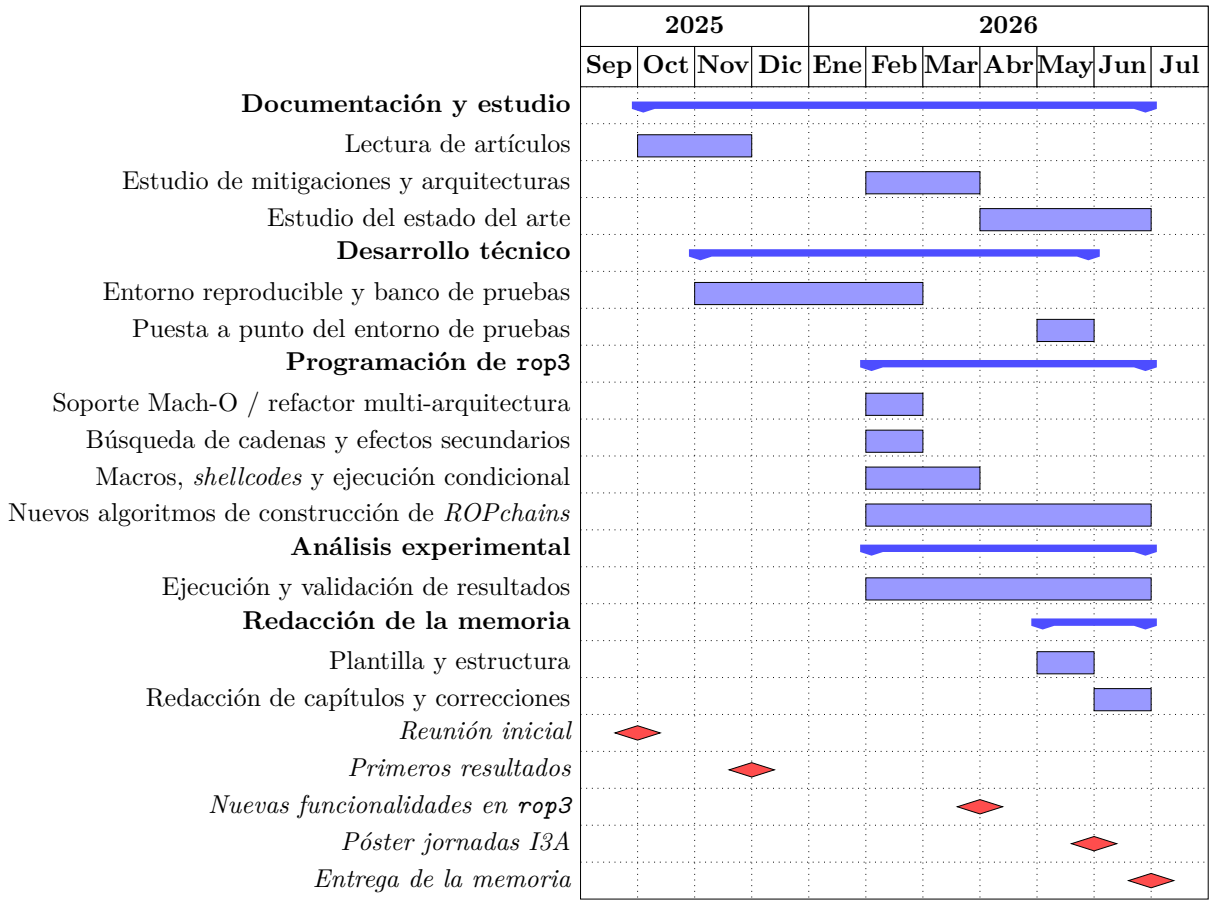

Figura A.1: Diagrama de Gantt con las fases e hitos principales del proyecto

Desarrollo técnico. Agrupa las tareas de infraestructura: la descarga reproducible de las bibliotecas de los distintos sistemas, la construcción del banco de pruebas, la instalación de la máquina virtual de macOS y la extracción de binarios Mach-O.

Programación. Núcleo del trabajo de implementación sobre la herramienta `rop3`, incluyendo el soporte para `Mach-O`, la mejora de los algoritmos de búsqueda de cadenas, la gestión de efectos secundarios y el refactor independiente de la arquitectura.

Análisis experimental. Ejecución de los experimentos, depuración de resultados anómalos y validación del comportamiento de la herramienta tras cada cambio relevante.

Redacción de la memoria. Última fase, dedicada a la creación de la plantilla, la planificación y la redacción y corrección de los distintos capítulos del documento.

A.3. Distribución mensual de la dedicación

La Tabla A.2 detalla las horas dedicadas cada mes a cada tarea. Permite apreciar cómo el trabajo evolucionó desde una fase inicial de estudio hacia el desarrollo y el análisis, concluyendo con la redacción del trabajo.

Mes	**Estud.**	**Técn.**	**Progr.**	**Anál.**	**Mem.**	**Reun.**	**Total**
Sep 2025	—	—	—	—	—	0,3	0,3
Oct 2025	5,0	—	—	—	—	0,3	5,3
Nov 2025	10,0	20,0	—	—	—	—	30,0
Dic 2025	—	—	—	—	—	0,3	0,3
Ene 2026	—	—	—	—	—	0,3	0,3
Feb 2026	5,0	22,0	17,0	10,0	—	0,9	54,9
Mar 2026	19,0	—	30,0	6,0	—	0,6	55,6
Abr 2026	16,5	—	—	8,0	—	—	24,5
May 2026	10,0	8,0	—	1,0	5,0	0,6	24,6
Jun 2026	10,0	—	15,0	1,0	90,0	—	116,0
Total	**75,5**	**50,0**	**62,0**	**26,0**	**95,0**	**3,3**	**311,8**

Tabla A.2: Horas dedicadas por mes y categoría de tarea