Universidad Zaragoza

Trabajo Fin de Grado en Ingeniería Informática

Búsqueda de conjuntos ROP Turing-completos en sistemas Windows

Daniel Uroz Hinarejos

Director: Ricardo J. Rodríguez

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Septiembre de 2016
Curso 2015/2016

Agradecimientos

Dedicar este trabajo de fin de grado
a mis padres por saber guiarme siempre que lo he necesitado,
a mi hermana por trasmitirme fuerza para nunca rendirme,
a Ricardo por su dedicación a esta oportunidad que me ha brindado.

Búsqueda de conjuntos ROP Turing-completos en sistemas Windows

RESUMEN

Los ataques *Return-Oriented Programming* (ROP) permiten la ejecución de un comportamiento arbitrario de un proceso sin la necesidad de introducir nuevo código en su espacio de memoria. El atacante organiza secuencias cortas de instrucciones ya existentes en su espacio de memoria que permiten controlar el flujo de una secuencia a la siguiente. Las secuencias de instrucciones son escogidas de manera que terminan en una instrucción "return", permitiendo así encadenar las secuencias almacenadas en pila. Esta terminación es la que da a la programación *return-oriented* su nombre.

La unidad de organización de la programación *return-oriented* es el *gadget*: una secuencia de instrucciones que, cuando se ejecutan, introduce un comportamiento bien definido, como realizar una o-exclusiva o una suma.

Una máquina de Turing es un modelo teórico capaz de resolver cualquier problema computacional mediante una serie de operaciones mínimas. Se dice que un sistema es Turing-completo si puede simular el comportamiento de una máquina de Turing, es decir, si puede realizar todas sus operaciones. Estas operaciones son, concretamente, realizar saltos condicionales, cargar una constante, mover un valor de una posición a otra, cargar un valor desde memoria, guardar un valor en memoria y realizar operaciones aritméticas y lógicas.

En este trabajo, se presenta una herramienta, llamada EasyROP, que tanto localiza los *gadgets* que implementan cada una de estas operaciones de Turing como automatiza la formación de ataques ROP. EasyROP permite buscar todas las operaciones para un binario dado, por lo que el atacante tiene la certeza de poder realizar cualquier computación si dicho binario se encuentra cargado en la memoria del proceso víctima. Finalmente, se han analizado los principales binarios de diferentes versiones del sistema operativo Windows (sobre arquitecturas Intel de 32 y 64 bits) para establecer la viabilidad de un ataque en estos sistemas.

Índice

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

Return-Oriented Programming (ROP) [Sha07, RBSS12] es una técnica que encadena conjuntos de instrucciones denominados *gadgets* que finalizan con una instrucción de cambio de flujo (concretamente, "`ret`"), los cuales ya se encuentran en el espacio de memoria de un proceso que está en ejecución (por ejemplo, la imagen de la biblioteca estándar *libc*). Esta instrucción "`ret`" permite transferir el control del programa a la dirección de retorno alojada en el tope de la pila, normalmente colocada por la instrucción ensamblador `call`. El encadenamiento de estos conjuntos permite la ejecución de un comportamiento arbitrario sin inyección de código gracias a vulnerabilidades software que permitan tomar el control de la pila del programa (como por ejemplo, vulnerabilidades de desbordamiento de búfer).

En arquitecturas de 32-bit, la pila es una región de la memoria del computador que permite guardar temporalmente las variables locales y los parámetros de las funciones, según el estándar de convención de llamada de x86 [Cor]. Además, contiene la dirección a la que retorna una función una vez ésta finalice. Se trata de una estructura LIFO (*Last In, First Out*), que cuenta con una operación `push` para introducir un dato en la pila y una operación `pop` que desapila un dato. Para controlar el flujo de la pila, el procesador cuenta con el registro `esp` que indica el tope de la pila en todo momento. La pila siempre crece hacia las direcciones de memoria menores.

```
1  void foo(char *str) {
2      char buf[10];
3      strcpy(buf, str);
4  }
5
6  void main(int argc, char **argv) {
7      foo(argv[1]);
8  }
```

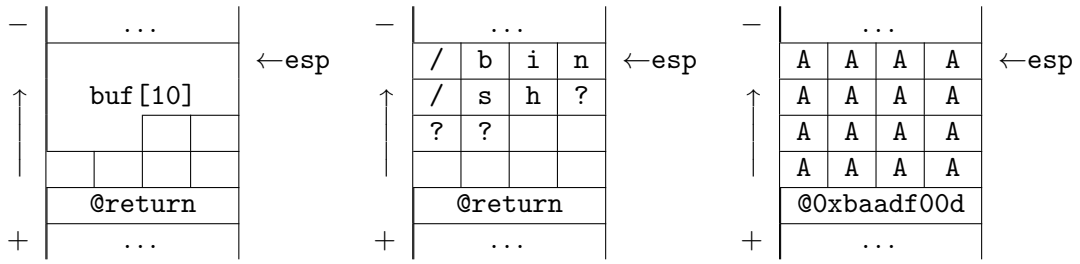
Código 1.1: Aplicación vulnerable.

```
1  _foo:
2      push ebp
3      mov ebp, esp
4      sub esp, 40
5      mov eax, DWORD PTR [ebp+8]
6      mov DWORD PTR [esp+4], eax
7      lea eax, [ebp-18]
8      mov DWORD PTR [esp], eax
9      call _strcpy
10     leave
11     ret
```

Código 1.2: Ensamblador de la función `foo`.

El Código 1.1 muestra un ejemplo de aplicación vulnerable por un ataque ROP. La

función `foo` declara una variable local `buf` de tamaño 10 (línea 2), la cual se traduce en espacio reservado en la pila del programa para almacenar temporalmente dicha variable (Código 1.2, línea 4). La vulnerabilidad se encuentra en la función `strcpy` de la línea 3 debido a que no comprueba el tamaño de la cadena origen que está copiando. Así, si la cadena proporcionada en `argv[1]` (parámetro de entrada del programa) es mayor que el tamaño de la variable `buf`, se sobrescriben las direcciones de memoria adyacentes. En la Figura 1.1 se muestra un ejemplo de ataque ROP mediante el desbordamiento de la pila, redirigiendo el código a la dirección `0xbaadf00d`.



(a) Variable local. (b) Sin desbordamiento. (c) Con desbordamiento.

Figura 1.1: Estado de la pila según la cadena copiada por `strcpy`.

Como se observa en la Figura 1.1c, desbordando el tamaño de la variable local se consigue sobrescribir la dirección a la que tiene que retornar el control una vez finalice la función. Si se sustituye dicha dirección por la dirección inicial del conjunto de *gadgets*, se consigue así modificar el flujo normal del programa. Esto permite encadenar conjuntos en la pila e ir ejecutando los *gadgets* deseados por el atacante, modificando así el comportamiento del programa a su voluntad.

Una máquina de Turing es un modelo teórico que puede resolver todos los problemas computacionales basándose en un conjunto de operaciones mínimas como sumar, restar, operaciones lógicas, comparaciones, etc. Se dice que un sistema es *Turing-completo* cuando puede simular el comportamiento de la máquina de Turing. Así pues, si un atacante cuenta con los suficientes *gadgets* para simular todos las operaciones de una máquina de Turing puede (por definición) resolver cualquier problema computacional y realizar así cualquier ataque en el sistema comprometido.

1.1. Objetivo

El objetivo de este trabajo es la creación de una herramienta para buscar *gadgets* que formen una máquina completa de Turing en sistemas Windows. Esta herramienta, llamada EasyROP, permitirá establecer si es viable o no un ataque ROP en las distintas versiones del sistema operativo Windows sobre arquitecturas Intel (32 y 64 bits). El proceso que se ha seguido es el siguiente:

- Estudiar los ataques ROP y sus variantes.

- Estudiar el modelo de la máquina universal de Turing.
- Estudiar los distintos desensambladores para arquitecturas x86 y x86_64.
- Establecer los gadgets necesarios para la creación de una máquina de Turing.
- Análisis, diseño, implementación y pruebas de la herramienta.
- Experimentación en las diferentes versiones del sistema operativo Windows.

Este proceso ha dado como resultado una herramienta implementada sobre el desensamblador Capstone. La herramienta está publicada bajo una licencia GNU GPL v3, se puede consultar su documentación y descargar su última versión a través de:

`https://github.com/uZetta27/EasyROP`

1.2. Organización

Este documento está organizado en 6 capítulos que muestran el contexto, desarrollo y alcance del Trabajo Fin de Grado (TFG) realizado.

En el Capítulo 2 se explican los ataques ROP en profundidad, se define y compara *frameworks* de desensamblado de binarios y, por último, se explica el modelo que sostiene la máquina universal de Turing. En el Capítulo 3 se presenta el diseño de los conjuntos ROP que forman una máquina universal de Turing en arquitecturas x86; y el análisis, diseño e implementación de la herramienta EasyROP. En el Capítulo 4 se presentan los resultados obtenidos con la utilización de la herramienta en diferentes sistemas Windows. El Capítulo 5 presenta otros trabajos relacionados con la búsqueda de conjuntos ROP, tanto estudios de viabilidad en otras plataformas como herramientas desarrolladas. Finalmente, en el Capítulo 6 se muestran las conclusiones del trabajo realizado y distintas líneas futuras.

A todos estos capítulos hay que añadir los siguientes apéndices:

- Apéndice A, que muestra un diagrama de Gantt con las horas dedicadas a las distintas partes del trabajo.
- Apéndice B: que detalla el catálogo completo de conjuntos de instrucciones ensamblador (de Intel x86) para las distintas operaciones con las que cuenta una máquina de Turing.

Capítulo 2

Conocimientos previos

En este capítulo se describe con más detalle el contexto sobre el que se desarrolla el TFG. Por un lado, se explica en profundidad la base sobre la que se sustentan los ataques ROP y sus características más significativas; y por otro lado, una explicación sobre la máquina universal de Turing.

2.1. Ataques *Return-Oriented Programming*

ROP fue propuesto por primera vez por Shacham en 2007 para arquitecturas Intel x86 [Sha07]. Desde entonces, su capacidad se ha extendido a un gran número de arquitecturas como Atmel AVR [FC08], SPARC [BRSS08], ARM [Kor09], Z80 [CFK+09], y PowerPC [Lin09].

Los ataques ROP suponen una evolución sobre la programación *return-into-libc* [Des97]: en vez de utilizar funciones completas de la biblioteca, se realiza el ataque mediante secuencias de instrucciones que finalizan con una instrucción de control de flujo como "`ret`". Una de sus características distintivas es que se pueden encontrar secuencias de instrucciones que no fueron generadas intencionadamente por el compilador. En arquitecturas Intel x86 esto es debido a que las instrucciones no están alineadas en memoria y la codificación de instrucciones es variable. Dicha característica permite eludir las defensas tomadas contra *return-into-libc*, como la eliminación de funciones o el cambio de ciertos comportamientos del compilador a la hora de generar código.

2.1.1. Bases de ROP

El primer paso para realizar un ataque ROP es descubrir alguna vulnerabilidad que permita derivar el flujo normal de programa. Tradicionalmente, se ha conseguido mediante un desbordamiento de pila [Ale96], donde el atacante sobrescribe la dirección de retorno de la pila a una dirección de su elección. Recuérdese que una de las funciones de la pila es almacenar la dirección de retorno de una función una vez ésta finalice. El segundo paso es la ejecución del propio ataque ROP.

Un ejecutable binario está formado por segmentos. Cada segmento contiene una secuencia de bytes que representa el código o los datos del programa. En el segmento `.text` se encuentran normalmente las instrucciones máquina que interpretará y ejecutará el proceso. El contador de programa (registro `eip`) marca cuál es la siguiente instrucción a ejecutar. ROP se ejecuta bajo el mismo principio pero haciendo uso de la pila, por lo que el contador de programa es simulado mediante el puntero de pila.

La unidad básica de los ataques ROP son los *gadgets*. Un *gadget* es un conjunto de instrucciones acabados normalmente en una instrucción `ret`. La ejecución de la instrucción `ret` al final del *gadget* supone que la doble palabra (32-bits de longitud) apuntada por `esp` (puntero de pila) se utilice como nuevo valor de `eip`; y se incremente `esp` en 4 bytes para apuntar a la siguiente doble palabra de la pila. En la Figura 2.1 se muestra la disposición de un ataque ROP en la pila del programa.

	...	
esp →	0x7c37638d	→ pop eax; pop ebx; ret
	0xdeadbeef	
	0xcafed00d	
	0x7c387f88	→ xor ecx, ecx; ret
	0x7c341591	→ neg ecx; ret
	0x7c367042	→ adc eax, ebx; ret
	0x7c34779f	→ pop ecx; ret
	0x5d345e7f	
	0x7c347f97	→ mov [ecx], eax; ret
	...	

Figura 2.1: Ejemplo de cadena ROP para ejecutar código a través de la pila.

2.1.2. Búsqueda de *gadgets*

La frecuencia con la que se pueden encontrar *gadgets* depende de las características de la arquitectura, lo que en [Sha07] se denomina *geometría*. En arquitecturas como MIPS las dobles palabras están alineadas a 32-bits. Esta característica supone que no haya equivocación respecto dónde empieza y termina una instrucción. Sin embargo, en arquitecturas como x86 las dobles palabras no están alineadas; es decir, aunque el compilador haya generado otras instrucciones, si se comienza las instrucciones uno o más bytes desplazados puede suceder que las nuevas instrucciones formen un *gadget*. Por ejemplo, si en la siguiente instrucción:

```
b8 89 41 08 c3        mov eax, 0xc3084189
```

se comienza un byte después, se obtiene:

```
89 41 08        mov [ecx+8], eax
c3              ret
```

Este comportamiento se produce gracias a que la arquitectura x86 tiene una gran densidad de instrucciones [Cor16], por lo que es bastante probable que un conjunto de bytes cualesquiera pueda ser interpretado como instrucciones válidas.

Una manera de encontrar *gadgets* es buscar una instrucción `ret` mediante su codificación en hexadecimal `c3`, y a partir de ahí recorrer hacia atrás n bytes hasta que se alcance un límite máximo de búsqueda. Esto supone que si se encuentra el *gadget* `a; b; c; ret` también se encuentren los *gadgets* `b; c; ret` y `c; ret`.

2.1.3. Ataques y defensas

Los ataques ROP superan las siguientes defensas extensamente adoptadas para mitigar las vulnerabilidades que se han ido descubriendo.

Write-XOR-eXecutable (W⊕X)

W⊕X asegura que las direcciones de memoria de la imagen de un proceso no puedan establecerse simultáneamente como de escritura (*Writable*) y ejecución (*eXecutable*). Esto se traduce en que el atacante no puede inyectar código que posteriormente pueda ser ejecutado. Tanto Linux a través del proyecto PaX [Tea03] como Microsoft Windows (a partir de XP SP2), OpenBSD y OS X implementan esta defensa tanto por software como por hardware si el procesador lo soporta, gracias a las tecnologías NX y XD de AMD e Intel, respectivamente.

Los ataques ROP evaden categóricamente esta defensa ya que no inyectan nuevo código, sino que inyectan direcciones a código legítimo que ya se encuentra en el espacio de memoria marcado como ejecutable.

Address-Space Layout Randomization (ASLR)

ASLR [Tea03] establece aleatoriamente en cada inicio del sistema operativo la localización de la pila, el *heap*, las bibliotecas dinámicas (por ejemplo, *libc*) y la imagen de proceso (por ejemplo, la sección `.text`). Esto dificulta los ataques *return-into-lib* y ROP, ya que dependen de ciertas direcciones de memoria de los procesos y no se sabe *a priori* dónde se localizarán.

En la práctica, sin embargo, ASLR sólo implementa la aleatorización de 16 de los 32 bits de la dirección base de cada biblioteca para no introducir una gran sobrecarga[1]. Esto supone que se pueda evitar esta medida mediante ataques de fuerza bruta. En [SPP+04] se demuestra que los ataques de fuerza bruta en plataformas de 32-bits comprometen los computadores en una media de 216 segundos. Así, se concluye que el único beneficio real de la implementación ASLR es aumentar el tiempo antes de que se realice el ataque (que no evitarlo totalmente).

[1]La dirección base es la dirección de memoria por defecto en la que se debe cargar un ejecutable en caso de que esté disponible. El resto de componentes del ejecutable son relativos a esta dirección

2.1.4. Evolución: *Jump-Oriented Programming*

Los ataques ROP han ganado gran popularidad desde que se propusieron, convirtiéndose en una de las principales técnicas de explotación en la actualidad. Esto supone que sus técnicas de defensa también hayan sido objeto de desarrollo.

Debido al uso desmedido de la instrucción `ret` por parte de los ataques ROP, varias técnicas de defensa se basan en la monitorización del número de ejecuciones con que cuenta dicha instrucción [DSW09, CXS+09]. Esto ha resultado en la búsqueda de técnicas que sean equivalentes a los ataques ROP pero sin la utilización de instrucciones `ret`.

Como respuesta, en [CDD+10] se presentó la sustitución de `ret` por el conjunto llamado *update-load branch* `pop x; jmp *x` (siendo `x` un registro de propósito general de la arquitectura), ya que cuenta con las mismas propiedades que las descritas en la Sección 2.1.1. Sin embargo, estas secuencias son más difíciles de encontrar, por lo que en vez de tratar de que cada secuencia de instrucciones acabe en un *update-load branch*, se puede reutilizar un único *update-load branch* como *trampolín* mediante un salto indirecto. Esto supone que en x86 sea necesario guardar la dirección de memoria del trampolín en un registro `z` y buscar que los gadgets acaben en la instrucción `jmp *z`, suponiendo que cuenta con una frecuencia suficiente de aparición.

2.2. *Frameworks* desensambladores

Un desensamblador permite transformar un determinado código binario en su correspondiente código ensamblador, transformándolo en un código mucho más legible. La mayoría de lenguajes ensamblador tienen una correspondencia directa entre los bytes y las instrucciones máquina, por lo que el proceso de desesamblado se basa en leer la cadena de bytes y realizar una traducción directa. Al no crear código de alto nivel, se pierde cierta información en el proceso, como símbolos, nombres de variables y macros.

Por otro lado, un *framework* desensamblador ofrece además interfaces para operar con el resultado que ofrecen mediante lenguajes de programación. Estas herramientas son una de las bases de la ingeniería inversa ya que permiten analizar código legítimo en busca de vulnerabilidades y también permiten un buen entendimiento del funcionamiento interno del código.

En la comparación de diferentes *frameworks* para el desarrollo de la herramienta, una característica indispensable ha sido el soporte de las arquitecturas x86 y x86_64. Como característica adicional, se ha buscado una buena documentación con códigos de ejemplo para agilizar el proceso de aprendizaje. Finalmente, se ha optado por realizar la herramienta en Python sobre Capstone [Comb] ya que Python es un lenguaje de propósito general muy utilizado en *scripting* que permite realizar prototipados rápidos, y Capstone es el *framework* que cuenta con la mejor documentación y el único con códigos de ejemplo que facilitan su comprehensión. En la Tabla 2.1 se muestra una comparación entre los distintos *frameworks* que se han considerado.

Características	Capstone	Distorm 3	BeaEngine	Udis86	Libopcode
Python	✓	✓	✓		
x86/x86_64	✓	✓	✓	✓	✓
Doc. completa	✓				
Códigos de ejemplo	✓				
Código abierto	✓	✓		✓	✓
Licencia	BSD	GPL	LGPL3	BSD	GPL

Tabla 2.1: Comparación de frameworks desensambladores.

Capstone Disassembly Framework

Capstone Disassembly Framework [Comb] es un *framework* muy potente implementado en C, pero con una sencilla API que se puede utilizar por más de una docena de lenguajes de programación. Desde su lanzamiento en el año 2013 ha sido incorporado en más de 170 productos, tales como IntelliJ IDEA o Radare2 [Comc].

Capstone cuenta con varias funciones que aportan distintos niveles de detalles para obtener las instrucciones que están codificadas en una cadena de bytes. Estas funciones son, en concreto:

- `disasm()`: devuelve una lista de objetos con su identificador, dirección, nombre, operandos, tamaño y bytes que forman la instrucción. El Código 2.1 muestra un ejemplo completo de su utilización en Python, mientras que en el Código 2.2 se observa su resultado.

- `disasm_lite()`: devuelve una tupla con los valores de dirección, tamaño, nombre y operandos. Este modo permite mejorar el rendimiento respecto a `disasm()` en un 30 % [Coma].

- `disasm()` con la opción `detail`: devuelve una lista de objetos con la misma información que `disasm()`, además de la información de los registros que intervienen implícitamente, el identificador del grupo al que pertenecen y detalles sobre los operandos de una operación, tales como: tipo de operando (registro, inmediato, número real o referencia a memoria con su desplazamiento), desplazamiento del operando, si la instrucción tiene una política de escritura aplazada (*write back*), el código de la condición si se trata de una comparación y si modifica las banderas. Este modo supone una gran sobrecarga por lo que sólo se debe de utilizar cuando sea necesario, pudiendo cambiar de modo en tiempo de ejecución.

```
1 from capstone import *
2
3 CODE = "\x55\x48\x8b\x05\xb8\x13\x00\x00"
4
5 md = Cs(CS_ARCH_X86, CS_MODE_64)
6 for i in md.disasm(CODE, 0x1000):
```

```
7       print "0x%x:\t%s\t%s" % (i.address, i.mnemonic, i.op_str)
```

Código 2.1: Ejemplo de utilización de Capstone en Python.

```
1  $ python test1.py
2
3  0x1000: push    rbp
4  0x1001: mov rax, qword ptr [rip + 0x13b8]
```

Código 2.2: Resultado del Código 2.1

2.3. Sistemas Turing-completos

Una maquina de Turing es un modelo teórico propuesto por Alan Turing [Tur36] que especifica que, dado una memoria y tiempo de ejecución infinitos, puede resolver cualquier problema computacional. Se dice que un sistema es *Turing-completo* si tiene la misma potencia computacional que una máquina de Turing, es decir, puede realizar las mismas operaciones. Las condiciones de disponer de memoria y tiempo infinitos no se pueden trasladar a un sistema físico real, por lo que no se tienen en cuenta a la hora de equiparar un sistema real con el modelo teórico.

Para resolver cualquier problema computacional, la máquina de Turing utiliza las operaciones de: (i) cargar una constante, (ii) mover un valor de una posición a otra, (iii) cargar un valor desde memoria, (iv) guardar un valor en memoria, (v) sumar y restar valores, (vi) realizar operaciones lógicas (tales como xor, and, or, not) y (vii) saltos condicionales.

La simulación de una máquina de Turing mediante *gadgets* se consigue agrupando los conjuntos de *gadgets* en operaciones. Cada operación contiene un conjunto de *gadgets* con la misma funcionalidad que las operaciones de la máquina de Turing, así que un solo conjunto es suficiente para implementar dicha funcionalidad.

Estos *gadgets* forman un lenguaje Turing-completo ya que los conjuntos propuestos contienen todas las instrucciones requeridas para implementar un OISC (*One-Instruction Set Computer*) [MP88], permitiendo así al atacante la ejecución de cualquier computación arbitraria.

Capítulo 3

Herramienta: EasyROP

Una vez analizados los ataques ROP y la máquina universal de Turing, el objetivo es crear una herramienta para buscar cada una de las operaciones necesarias para que un ataque ROP sea Turing-completo, permitiendo así que un atacante tenga la certeza de que puede ejecutar todo ataque que desee.

3.1. Conjuntos ROP Turing-completos

Para que un atacante tenga la certeza de que puede realizar cualquier computación arbitraria, debe encontrar para cada operación de la máquina de Turing al menos un conjunto de *gadgets* equivalente. Es decir, tendría que encontrar conjuntos para:

- Cargar una constante en un registro (**lc**).
- Mover un valor de un registro a otro (**move**).
- Cargar desde memoria (**load**).
- Guardar en memoria (**store**).
- Sumar y restar valores (**add**, **sub**).
- Realizar operaciones lógicas (**xor**, **and**, **or**, **not**).
- Comparar dos valores y saltar a otra dirección según el resultado (**cond1**, **cond2**).

Las operaciones lógicas necesarias para conseguir una máquina Turing-completa son and, or, xor y not; pero gracias a las leyes de De Morgan se pueden simular todas únicamente con una operación and/or y una operación xor/not/neg, en caso de que sea necesario.

La realización de una operación puede ser inmediata por encontrar *gadgets* con las instrucciones adecuadas, pero también pueden ser necesarias varias instrucciones en un orden específico para llevar a cabo la operación. A esta agrupación de instrucciones se le ha denominado *conjunto*. Esto ha supuesto que se haya tenido que especificar diferentes

formas para realizar cada una de las operaciones. Se puede acceder al catálogo completo de conjuntos en el Anexo B. A su vez, cada conjunto se puede dividir en un *gadget* por cada una de sus instrucciones. Esta agrupación de *gadgets* se ha denominado *ROP chain*.

En la Figura 3.1 se muestra un ejemplo de la operación de Turing **move**, compuesta por cuatro conjuntos y cada uno de ellos compuesto a su vez por una serie de instrucciones. Los dos últimos conjuntos son ejemplos de *ROP chains*.

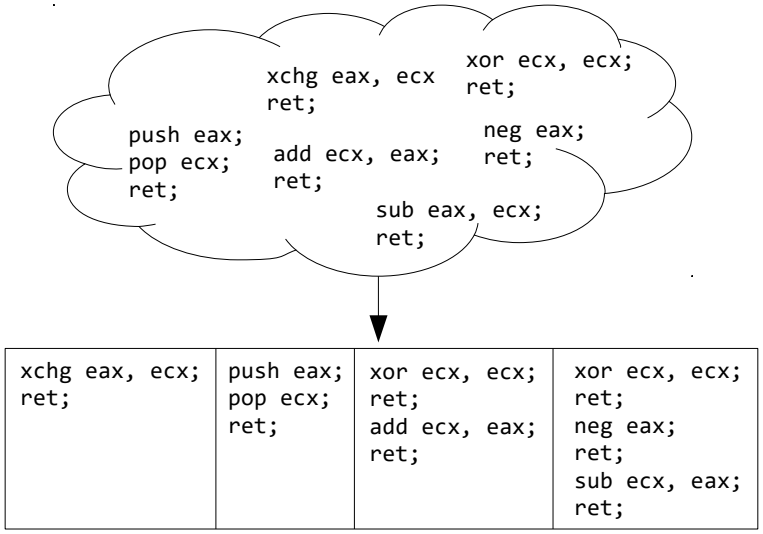


Figura 3.1: Conjuntos para mover el valor del registro `eax` a `ecx`.

Saltos condicionales

La realización de un salto condicional en los ataques ROP son operaciones más complejas, por lo que se han dividido en dos tareas **cond1** y **cond2**. No se pueden utilizar las operaciones de salto típicas como `jmp` y sus variantes debido a que afectan al contador de programa. Como se propuso en [Sha07], para realizar un salto condicional es necesario transferir el valor de una bandera a un registro de propósito general después de realizar la comparación (**cond1**) y utilizar dicho registro para modificar el valor de `esp` con el desplazamiento deseado para el salto (**cond2**).

La primera tarea **cond1** tiene como objetivo saber si un valor es mayor que otro. La instrucción de resta `sub` establece la CF (*Carry Flag* o bandera de acarreo) cuando el sustraendo es mayor que el minuendo. De esta forma, se obtiene que un valor es mayor que otro realizando una resta entre ambos. El siguiente paso es transferir el valor de CF a un registro de propósito general. La instrucción de suma `adc` realiza una suma de dos operandos añadiendo el valor de CF al resultado. Así, si el valor de los dos operandos que intervienen en la operación es 0, se obtiene en el registro de destino un 0 ó 1 en función del valor de CF.

La segunda tarea **cond2** tiene como objetivo modificar el valor de `esp` con el desplazamiento deseado para el salto. Tras **cond1**, se tiene un registro con el valor 0 ó 1 de la CF y se debe transformar en 0 o *esp_delta*, siendo *esp_delta* el valor de desplazamiento que se quiere aplicar a `esp`. La instrucción `neg` calcula el complemento a dos de un registro, por lo que si el registro que contiene el valor de CF es un 1, se convierte en el patrón todo 1; y si contiene un 0, se convierte en el patrón todo 0. Así, ejecutando una operación `and` entre el resultado y *esp_delta* se obtiene 0 o *esp_delta*. Ahora sí, con el valor de desplazamiento se puede modificar el valor de `esp` simplemente con una operación de suma. En la Figura 3.2 se muestra el ejemplo completo de un salto condicional.

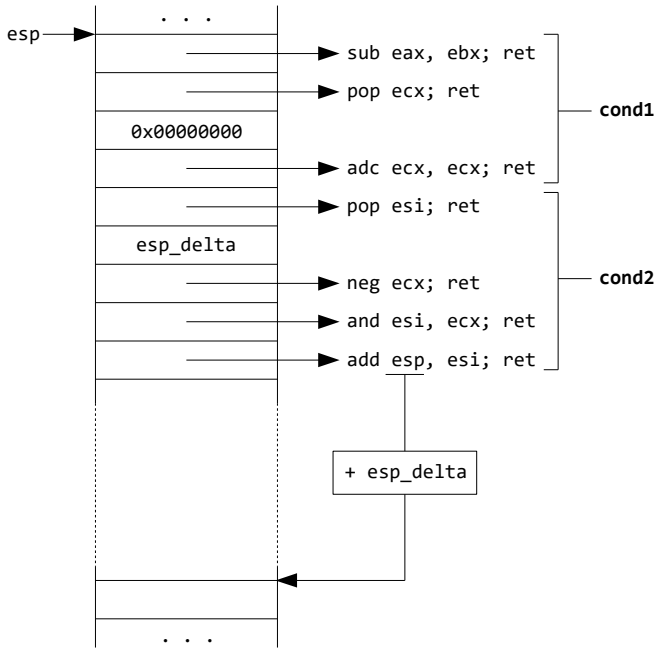


Figura 3.2: Salto condicional siendo $ebx > eax$.

3.2. Análisis

Esta fase permite determinar el alcance de la herramienta, para ello se han establecido los siguientes requisitos funcionales y no funcionales:

Requisitos funcionales

RF1 El usuario podrá visualizar todos los gadgets existentes en un binario de Windows (Portable Executable – PE) [Fre].

RF2 El usuario podrá modificar el tamaño en bytes de los gadgets a buscar.

RF3 El usuario podrá deshabilitar la búsqueda de gadgets que no finalicen en `ret`; es decir, los acabados en saltos a otro segmento de código (`retf`) o cambios de flujo incondicionales (`jmp`, `call`).

RF4 El usuario podrá buscar una operación, tanto de la máquina de Turing como especificada por el usuario.

RF5 El usuario podrá especificar el registro fuente y/o destino en la búsqueda de una operación.

RF6 El usuario podrá habilitar la búsqueda de operaciones mediante la creación de cadenas formadas por una sucesión de gadgets, denominadas *ROP chains*.

RF7 El usuario podrá saber si un binario cuenta con los suficientes gadgets para formar una máquina de Turing.

RF8 El usuario podrá saber si un sistema operativo cuenta con los suficientes gadgets en sus binarios para formar una máquina de Turing.

RF9 El usuario podrá automatizar la formación de ataques ROP especificándolos a través de secuencias de operaciones.

Requisitos no funcionales

RNF1 La herramienta debe funcionar al menos en sistemas Windows, Linux y OS X.

RNF2 La herramienta debe facilitar el soporte de distintas fuentes para la búsqueda de operaciones (XML, bases de datos, etc).

RNF3 La herramienta debe facilitar la extensión a binarios de otras arquitecturas.

RNF4 La herramienta debe facilitar la extensión de operaciones especificadas por los usuarios.

3.3. Diseño

Una vez llevado a acabo el análisis de las funcionalidades, es necesario llevar a cabo un diseño que permita cumplirlas. En la Figura 3.3 se muestra una vista de módulos de la herramienta donde se refleja las clases que la componen y sus relaciones.

Para cumplir con los requisitos **RNF2** y **RNF3**, se ha optado por el patrón de diseño estructural *Facade*. Un patrón de diseño es una solución plantilla a un problema recurrente. *Facade* permite proporcionar una interfaz de alto nivel unificada a un conjunto de objetos dentro de un subsistema, permitiendo abstraerse de los detalles internos.

A continuación, se detallan las clases más relevantes:

- **Core**: es la clase principal de la herramienta. Se encarga de realizar la búsqueda de *gadgets*, de operaciones y de *ROP chains* en los binarios.

- **RopGenerator**: es la clase encargada de automatizar la formación de los ataques ROP. Lee un fichero con la secuencia de operaciones y utiliza la clase **Core** para la búsqueda de los *gadgets* equivalentes a estas operaciones.
- **Binary**: esta interfaz especifica las operaciones básicas que debe implementar cualquier binario a analizar. Su incorporación permite que sea transparente la estructura del binario para la clase **Core**, permitiendo a su vez que se puedan extender las arquitecturas soportadas por la aplicación simplemente extendiendo esta interfaz y realizando el análisis sintáctico oportuno para obtener las características principales de los binarios en esas nuevas arquitecturas:
  - `file_name`: nombre del fichero binario.
  - `raw_bytes`: secuencia de bytes que compone el binario.
  - `entry_point`: dirección en la que empieza el código.
  - `exec_sections`: lista con todos los bytes de la secciones que contienen código.
  - `arch`: arquitectura del binario (x86, ARM, PowerPC, etc).
  - `arch_mode`: si se trata de un binario de 32-bits o de 64-bits.
- **Parser**: esta interfaz especifica las operaciones a implementar por cualquier clase que tenga como finalidad obtener todos los conjuntos de *gadgets*. Incorporando clases que implementen esta interfaz se conseguirá añadir otras fuentes para la obtención de conjuntos, como bases de datos, servicios REST, etc.
- **Clases Operation-Set-Instruction**: estas tres clases representan los conjuntos de Turing explicados en la Sección 3.1. Esta jerarquía establece que cada operación está compuesta por una lista de conjuntos, mientras que cada conjunto está formado a su vez por una lista de instrucciones.

  Las tres clases cuentan con las funciones `set_dst()` y `set_src()` para establecer el registro de destino y el registro de origen de una operación. Por ejemplo, establecer el destino y el origen para la operación de mover de un registro a otro es trivial para el conjunto `xchg src, dst;` pero no para un conjunto más complejo como `xor dst, dst; neg src; sub dst, src`.

3.4. Generación de contenido por parte del usuario

Para la especificación de los conjuntos se ha optado por ficheros XML debido a que permiten estructurar la información y que cuentan con soporte para la mayoría de lenguajes de programación, al ser ampliamente utilizados en la actualidad. Además, para establecer una estructura fija durante la creación de los XML se ha definido un *Document Type Definition* (DTD) que establece una sintaxis formal para definir el conjunto específico de etiquetas y sus relaciones específicas. De esta forma, se puede controlar que la creación de datos por parte de los usuarios siga la estructura propuesta. Un ejemplo de la formación de un XML con su respectivo DTD se muestra en el Código 3.1

Un fichero XML permite especificar una serie de operaciones, formadas cada una de ellas por conjuntos de instrucciones. Cada instrucción está compuesta por 0, 1 ó 2 registros:

- `dst`: registro destino.
- `src`: registro fuente.
- `aux`: registro auxiliar (sólo se puede especificar uno por conjunto o *set*).
- `[dst]`: dirección de destino alojada en un registro.
- `[src]`: dirección de fuente alojada en un registro.
- `{eax, ebx, ecx...}`: registro específico de propósito general.
- `[{eax, ebx, ecx...}]`: dirección alojada en un registro específico de propósito general.
- `<reg{1,2} value ="0xFFFFFFFF">`: valor obligatorio del registro.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE operations [
        <!ELEMENT operations (operation)+>
        <!ELEMENT operation (set)+>
        <!ATTLIST operation
                name CDATA #REQUIRED>
        <!ELEMENT set (ins)+>
        <!ELEMENT ins (reg1|reg2)*>
        <!ATTLIST ins
                mnemonic CDATA #REQUIRED>
        <!ELEMENT reg1 (#PCDATA)>
        <!ATTLIST reg1
                value CDATA #IMPLIED>
        <!ELEMENT reg2 (#PCDATA)>
        <!ATTLIST reg2
                value CDATA #IMPLIED>
        ]>
<operations>
    <operation name="move">
        <set>
            <ins mnemonic="xor">
                <reg1>dst</reg1>
                <reg2>dst</reg2>
            </ins>
            <ins mnemonic="add">
                <reg1>dst</reg1>
                <reg2>src</reg2>
            </ins>
        </set>
    </operation>
</operations>
```

Código 3.1: Fichero XML con su DTD correspondiente para formar una operación con un conjunto.

Para la generación de ataques ROP, es necesario un fichero de texto en el que se especifiquen todas las operaciones para completar el ataque. En cada línea se establece una operación (definida en un fichero XML con la estructura del Código 3.1) y sus registros implicados. Estos registros se pueden especificar tanto con máscaras (`regX`) como con registros específicos de propósito general. En el Código 3.2 se muestra un ejemplo para encadenar operaciones de un ataque ROP.

```
1 lc(reg1)
2 lc(reg2)
3 sub(reg2, reg1)
4 lc(reg3)
5 store(reg3, reg2)
```

Código 3.2: Fichero para la generación de ataques ROP.

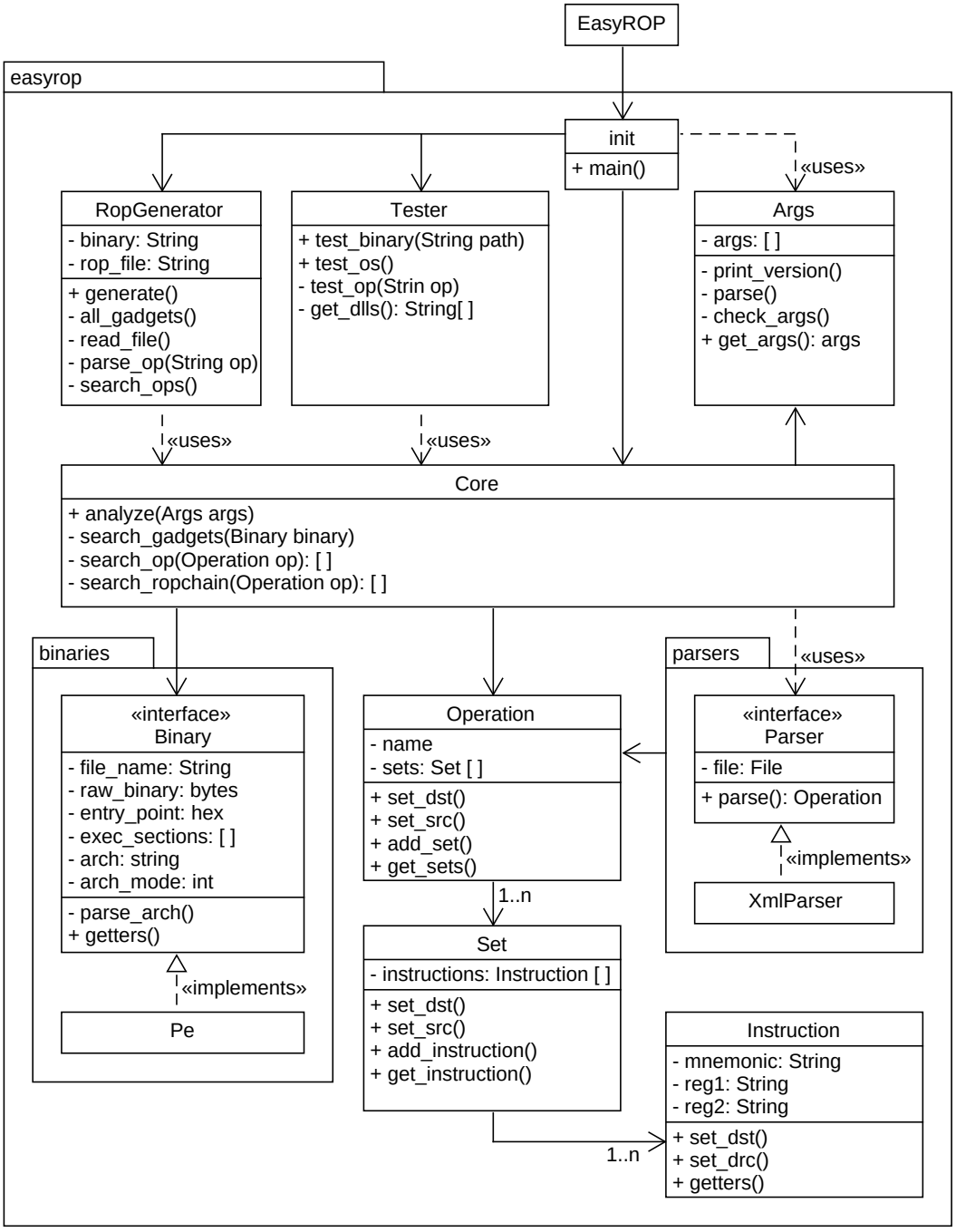


Figura 3.3: Vista de módulos de la herramienta EasyROP.

Capítulo 4

Experimentación en sistemas Windows

Una vez creada la herramienta, se procede a la realización de una batería de pruebas para determinar si distintas versiones del sistema de operativo Windows cuentan con la cantidad de *gadgets* suficientes para implementar cada una de las operaciones de una máquina de Turing. Estas pruebas se han realizado mediante un script en Python que tiene en cuenta las cuestiones discutidas a continuación.

4.1. Preparación de pruebas

Antes de establecer el diseño de las pruebas hay que especificar cuáles son los binarios a analizar. Recuérdese que los ataques ROP se aprovechan de las imágenes de los procesos que están en el espacio de memoria de un ejecutable. Debido a esta característica, la biblioteca *libc* ha sido objetivo de distintos ataques en sistemas Linux [Des97, Sha07] ya que siempre se encuentra en memoria para el funcionamiento de cualquier binario. En el caso de Windows, las bibliotecas de enlace dinámico (*dynamic-link libraries* o DLL) son los ejecutables cargados en memoria por programas en ejecución para realizar funciones comunes.

Windows cuenta con multitud de DLL, tanto para su propio funcionamiento como para el correcto funcionamiento de cualquier ejecutable. Por tanto, se ha reducido el rango de DLL para analizar a aquellas que cuenten con una probabilidad muy alta de ser utilizadas por cualquier binario. Las referencias a las DLL propias del sistema operativo y más utilizadas se encuentran en el objeto del sistema operativo KnownDLLs. El valor de este objeto se encuentra en el registro de Windows, concretamente en la clave `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\KnownDLLs`.

Así, se ha optado por un subconjunto de 14 DLL bajo la carpeta `\system32` (carpeta del sistema), pertenecientes a KnownDLLs. Estas DLL permiten a los ejecutables la utilización de funciones básicas y avanzadas del sistema operativo, servicios del *shell*, creación de interfaces gráficas, utilización de dispositivos de entrada/salida, etc. Este

subconjunto está formado por las DLL: *kernel32*, *advapi32*, *gdi32*, *comdlg32*, *shell32*, *shlwapi*, *msvcrt*, *ole32*, *psapi*, *rpcrt4*, *setupapi*, *user32*, *wldap32* y *ws2_32*. En el caso particular de Windows XP, algunas DLL de este subconjunto se encuentran en la carpeta `\system32` pero no pertenecen al listado de DLL de KnownDLLs, así que no se han considerado.

Una vez establecidas las DLL se procede al diseño de las pruebas. Simplemente encontrar *gadgets* para cada una de las operaciones de Turing no es suficiente, sino que también hay que comprobar la relación necesaria para encadenar unas operaciones con otras.

Si se observa un programa ordinario cualquiera, es necesaria una forma de definir variables y asignarles un valor para después operar con ellas, ya sea mediante operaciones lógicas, comparaciones, llamadas a funciones, etc. Poniendo como ejemplo Java, si en un primer momento no hay forma de inicializar la variable `int i = 0`, es imposible que más adelante se pueda operar con ella para establecer si es mayor que la longitud de un vector o incrementar en uno su valor. El mismo principio se ha supuesto aquí: si en un primer momento no se puede inicializar la variable mediante la operación de cargar una constante, no puede intervenir más adelante en operaciones de tipo lógico, ni aritméticas, etc.

Así, el primer paso es saber qué registros intervienen en la operación de cargar una constante y después comprobar una por una el resto de operaciones que forman una máquina de Turing teniendo en cuenta la intervención de dichos registros. Si por ejemplo se descubre que el registro `eax` puede soportar la operación de carga de una constante, la búsqueda del resto de operaciones se realizará mediante dicho registro con cualquier tamaño, es decir: `eax` (32-bits), `ax` (16-bits), `ah` (parte alta de `ax`, 8-bits) y `al` (parte baja de `ax`, 8-bits). Si no se encuentran todas las operaciones, el binario analizado no puede formar una máquina de Turing ya que todas las operaciones definidas son indispensables.

4.2. Entorno de pruebas

Una vez especificadas las pruebas se ha procedido con su ejecución en distintas versiones del sistema operativo. La instalación de los sistemas se ha realizado mediante una instalación en limpio en máquinas virtuales de Oracle VirtualBox con 4GB de RAM y 32 GB de espacio libre. El equipo anfitrión está compuesto por un Intel Core i7-3610QM a 2,3 GHz con 8GB DDR3-1600 MHz y 256GB de SSD. Los sistemas operativos que se han probado son, en particular:

- Microsoft Windows XP Professional 5.1.2600 Service Pack 3 Build 2600 (32-bit);
- Microsoft Windows XP Professional x64 Edition 5.2.3790 Service Pack 2 Build 3790 (64-bit);
- Microsoft Windows 7 Professional 6.1.7601 Service Pack 1 Build 7601 (32/64-bit);
- Microsoft Windows 8.1 Pro 6.3.9600 Build 9600 (32/64-bit);
- Microsoft Windows 10 Education 10.0.14393 Build 14393 (32/64-bit).

4.3. Discusión de resultados

Se han buscado todas las operaciones de la máquina de Turing en cada una de las DLL consideradas de los sistemas operativos. La búsqueda de cada operación se ha realizado mediante sus conjuntos y, si no se han encontrado coincidencias, mediante *ROP chains*. El tiempo total de experimentación ha sido de 11h12m. En la Tabla 4.1 se muestra un desglose por sistema operativo y versión.

Versión	**32-bit**	**64-bit**
Windows XP	53m	1h02m
Windows 7	1h20m	2h52m
Windows 8.1	1h09m	1h11m
Windows 10	1h27m	1h13m

Tabla 4.1: Tiempo de pruebas en cada versión del sistema operativo.

Los resultados detallados de la experimentación se exponen en las Tablas 4.3—4.10. Cada fila de una tabla representa las operaciones de Turing encontradas por DLL, siendo la última fila un resumen de las operaciones disponibles considerando todas las DLL analizadas.

En vistas generales, los resultados de la experimentación son favorables. Existen suficientes *gadgets* para formar una máquina de Turing en 5 de las 8 versiones analizadas (Tabla 4.2). Esto supone que un atacante puede ejecutar cualquier código que desee en un 59 % del mercado, según estadísticas de agosto de 2016 acorde a [W3S].

Además, cabe destacar que la DLL *shell32* es la responsable de poder crear una máquina de Turing en dichos 5 sistemas operativos. Por tanto, es una perfecta candidata a la hora de formar ataques ROP. La siguiente candidata es *rpcrt4*, que permite realizar todas las operaciones menos saltos condicionales en casi todos los sistemas operativos.

Aunque existen 3 sistemas operativos que no pueden formar todas las operaciones de una máquina de Turing ya que les faltan gadgets para formar saltos condicionales, sí que pueden formar el resto de operaciones. Esto supone que igualmente el atacante adquiere suficiente poder, a pesar de no ser capaz de realizar cualquier computación arbitraria, situando así la cantidad de equipos vulnerables en un 78 %.

Versión	**32-bit**	**64-bit**
Windows XP	✗	✗
Windows 7	✓	✗
Windows 8.1	✓	✓
Windows 10	✓	✓

Tabla 4.2: Formación de una máquina de Turing en cada versión del sistema operativo.

DLL	lc	load	store	add	sub	xor	and	or	not	cond1	cond2	move
advapi32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
comdlg32	✓	✓	✓	✓			✓	✓				✓
gdi32	✓	✓	✓	✓	✓	✓		✓	✓			✓
kernel32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
ole32	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
rpcrt4	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
shell32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
user32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
wldap32	✓		✓		✓							✓
GLOBAL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓

Tabla 4.3: Detalle de las DLL de Windows XP (32-bits).

DLL	lc	load	store	add	sub	xor	and	or	not	cond1	cond2	move
advapi32	✓	✓	✓	✓	✓		✓	✓				✓
comdlg32	✓	✓	✓	✓			✓	✓				✓
gdi32	✓	✓	✓	✓	✓	✓		✓	✓			✓
kernel32	✓	✓	✓	✓	✓		✓	✓				✓
ole32	✓	✓	✓	✓	✓	✓			✓			✓
rpcrt4	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
shell32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
user32	✓	✓		✓	✓		✓	✓				✓
wldap32	✓			✓	✓							✓
GLOBAL	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓

Tabla 4.4: Detalle de las DLL de Windows XP (64-bits).

DLL	lc	load	store	add	sub	xor	and	or	not	cond1	cond2	move
advapi32	✓	✓	✓	✓	✓	✓		✓	✓			✓
comdlg32	✓	✓	✓	✓			✓	✓				✓
gdi32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
kernel32	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
msvcrt	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
ole32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
psapi												
rpcrt4	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
setupapi	✓	✓	✓	✓	✓			✓				✓
shell32	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
shlwapi	✓		✓	✓	✓		✓	✓	✓			✓
user32	✓	✓	✓	✓	✓		✓	✓				✓
wldap32	✓	✓		✓	✓	✓		✓				✓
ws2_32	✓	✓	✓	✓	✓	✓		✓	✓			✓
GLOBAL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Tabla 4.5: Detalle de las DLL de Windows 7 (32-bit).

DLL	lc	load	store	add	sub	xor	and	or	not	cond1	cond2	move
advapi32	✓	✓	✓	✓	✓			✓				✓
comdlg32	✓	✓	✓	✓			✓					✓
gdi32	✓	✓		✓				✓				✓

DLL	lc	load	store	add	sub	xor	and	or	not	cond1	cond2	move
kernel32	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
msvcrt	✓	✓	✓	✓	✓		✓			✓		✓
ole32	✓	✓	✓	✓	✓	✓		✓	✓			✓
psapi	✓											
rpcrt4	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
setupapi	✓		✓	✓	✓							✓
shell32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
shlwapi	✓	✓		✓	✓		✓					✓
user32	✓	✓		✓	✓			✓				✓
wldap32	✓											✓
ws2_32	✓	✓	✓	✓				✓				✓
GLOBAL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓

Tabla 4.6: Detalle de las DLL de Windows 7 (64-bit).

DLL	lc	load	store	add	sub	xor	and	or	not	cond1	cond2	move
advapi32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
comdlg32	✓	✓	✓	✓	✓	✓	✓		✓			✓
gdi32	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
kernel32	✓	✓	✓	✓			✓	✓				✓
msvcrt	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
ole32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
psapi												
rpcrt4	✓	✓	✓	✓		✓	✓	✓	✓			✓
setupapi	✓	✓	✓	✓	✓		✓	✓	✓			✓
shell32	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
shlwapi	✓	✓	✓	✓	✓	✓		✓	✓			✓
user32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
wldap32	✓	✓	✓		✓							✓
ws2_32	✓	✓	✓	✓				✓				✓
GLOBAL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Tabla 4.7: Detalle de las DLL de Windows 8.1 (32-bit).

DLL	lc	load	store	add	sub	xor	and	or	not	cond1	cond2	move
advapi32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
comdlg32	✓	✓	✓	✓	✓	✓	✓		✓			✓
gdi32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
kernel32	✓	✓	✓	✓	✓		✓	✓		✓		✓
msvcrt	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
ole32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
psapi												
rpcrt4	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
setupapi	✓	✓	✓	✓	✓		✓	✓	✓			✓
shell32	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
shlwapi	✓	✓	✓	✓	✓	✓		✓	✓			✓
user32	✓	✓	✓	✓	✓		✓	✓	✓			✓
wldap32	✓	✓	✓		✓							✓
ws2_32	✓	✓	✓	✓			✓	✓				✓
GLOBAL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Tabla 4.8: Detalle de las DLL de Windows 8.1 (64-bit).

DLL	lc	load	store	add	sub	xor	and	or	not	cond1	cond2	move
advapi32	✓	✓	✓	✓		✓	✓		✓			✓
comdlg32	✓	✓	✓	✓	✓		✓	✓				✓
gdi32	✓		✓	✓								
kernel32	✓	✓	✓	✓	✓		✓					✓
msvcrt	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
ole32	✓	✓	✓	✓	✓			✓				✓
psapi												
rpcrt4	✓	✓	✓	✓		✓	✓	✓	✓			✓
setupapi	✓		✓	✓								✓
shell32	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
shlwapi	✓	✓	✓	✓	✓			✓				✓
user32	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
wldap32	✓	✓	✓									✓
ws2_32	✓	✓	✓	✓		✓		✓	✓			✓
GLOBAL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Tabla 4.9: Detalle de las DLL de Windows 10 (32-bit).

DLL	lc	load	store	add	sub	xor	and	or	not	cond1	cond2	move
advapi32	✓	✓	✓	✓		✓	✓		✓			✓
comdlg32	✓	✓	✓	✓	✓		✓	✓				✓
gdi32	✓			✓								✓
kernel32	✓	✓	✓	✓	✓		✓	✓				✓
msvcrt	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
ole32	✓	✓	✓	✓	✓			✓				✓
psapi												
rpcrt4	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓
setupapi	✓		✓	✓								✓
shell32	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
shlwapi	✓	✓	✓	✓	✓			✓				✓
user32	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
wldap32	✓	✓	✓									✓
ws2_32	✓	✓	✓	✓		✓		✓	✓			✓
GLOBAL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Tabla 4.10: Detalle de las DLL de Windows 10 (64-bit).

Capítulo 5

Trabajo relacionado

En esta sección se presentan los trabajos relacionados con el TFG, tanto trabajos de investigación como herramientas dispuestas al público.

Búsqueda de gadgets

En el artículo original donde se proponen los ataque ROP [Sha07], se especifica un conjunto de *gadgets* escogidos a mano tras un análisis detallado de la biblioteca *libc*. Estas combinaciones de *gadgets* son complejas, debido a que no existen versiones simplificadas con lo que si se encuentra una versión diferente de la biblioteca será necesario realizar un nuevo análisis para encontrar las combinaciones de *gadgets*.

En [HHF09] se propone una herramienta que automatiza la búsqueda de *gadgets* especificando una serie de instrucciones para formar cada una de las operaciones de Turing, pero dicha especificación está limitada a una instrucción por *gadget*.

En [HSL+12] se especifica una serie de *microgadgets* (*gadgets* de longitud de 2 ó 3 bytes) que permiten implementar una máquina de Turing. Al tratarse de *gadgets* de menor tamaño al habitual, es más probable que se encuentren. Se prueba su viabilidad en varias distribuciones de Linux y diferentes navegadores web mediante una herramienta en Python, dando resultados positivos en las 8 distribuciones analizadas.

Respecto a trabajos centrados en Windows, no hay disponible ningún trabajo en el que se analice en detalle la viabilidad de los ataques ROP.

Herramientas ROP

ROPgadget Tool [Sal] es una herramienta creada en Python para obtener los *gadgets* en un binario. Sus características más reseñables son: buscar un identificador de instrucción específico en segmentos marcados como ejecución, buscar cadenas en segmentos marcados como lectura, buscar entre dos direcciones dadas, buscar mediante expresiones regulares, crear una cadena ROP de ejemplo con los *gadgets* encontrados para realizar una llamada a una función y ofrece soporte para los formatos de fichero ELF, PE, Mach-O en arquitecturas x86, x64, ARM, ARM64, PowerPC, SPARC y MIPS.

Ropper [Sch] es también una herramienta creada en Python que cuenta con unas características muy similares a las de ROPgadget Tool, diferenciándose en que además permite crear cadenas ROP para realizar distintas ejecuciones:

- *execve*: permite ejecutar un proceso en Linux, por ejemplo la creación de un *shell* mediante `/bin/sh`.
- *mprotect*: permite cambiar la protección de una región de memoria en Linux, pudiendo marcarla como inaccesible, lectura, escritura o ejecución.
- *virtualprotect*: permite cambiar la protección de una región de memoria en Windows.

La herramienta EasyROP se desmarca de ambas ofreciendo al usuario una forma de generar automáticamente ataques ROP mediante el encadenamiento de operaciones compuestas por una sucesión de *gadgets*. Además, EasyROP permite a los usuarios definir sus propias operaciones complejas, establecer cuáles son los registros de origen y/o destino para la búsqueda de operaciones, la búsqueda de operaciones mediante la formación de cadenas de *gadgets* o de *ROP chains* y, por último, establecer si es viable la creación de cualquier ataque ROP mediante el análisis de un binario dado o un sistema operativo.

Contribución

Mientras que las herramientas existentes se centran en descubrir todos los *gadgets* que se pueden formar a partir del binario de un programa, EasyROP forma con todos los *gadgets* las operaciones necesarias para realizar un ataque. Al establecer este paso intermedio, se consigue un capa de abstracción que permite centrarse en el objetivo del ataque, ya que la herramienta forma todas las operaciones de alto nivel necesarias para llevarlo a cabo.

Además, esta herramienta une las dos vertientes expuestas para la búsqueda de *gadgets* que implementen cada una de las operaciones de la máquina de Turing, pudiendo añadir cualquier usuario nuevos conjuntos (tanto equivalentes a operaciones de la máquina de Turing como no equivalentes) mediante la sintaxis propuesta en la Sección 3.4. La búsqueda de operaciones incluye además la posibilidad de creación automática de cadenas de *gadgets*.

Por otra parte, también aporta un análisis detallado de la viabilidad de un ataque ROP en distintos sistemas Windows, indicando cuáles son las operaciones que se pueden encontrar en cada una de las DLL de Windows analizadas.

Capítulo 6

Conclusiones

En este último capítulo se presentan las conclusiones del trabajo, sus implicaciones y posibles líneas de trabajo futuro.

Los ataques ROP son una técnica de ataque que consigue evitar las defensas implementadas por los sistemas actuales, como W⊕X y ASLR. Si a esta característica se le suma la capacidad de imitar una máquina de Turing, los ataques ROP son una herramienta muy potente para ejecutar cualquier computación arbitraria o permitir la fuga de información.

El trabajo propuesto se centra en complementar los ataques ROP con la formación de conjuntos Turing-completos. Esta línea de trabajo ha permitido contribuir mediante la creación de una herramienta, llamada EasyROP, capaz de buscar todas las operaciones de Turing en un binario mediante distintos conjuntos de *gadgets*. Además, se permite la creación automática de ataques ROP, y la ampliación de conjuntos y operaciones por parte de los usuarios finales de la herramienta.

Mientras que la gran mayoría de estudios de ataques ROP se han centrado en sistemas Linux, el uso de EasyROP ha permitido también contribuir con el estudio de viabilidad de estos ataques en sistemas Windows. Este estudio se ha centrado en localizar conjuntos Turing-completos en cada una de las versiones del sistema operativo desde Windows XP y arquitecturas Intel, tanto en 32 como en 64 bits. Al analizar el sistema operativo más extendido entre los ordenadores personales, se ha comprobado que se puede realizar cualquier ataque ROP en el 59 % del mercado actual.

Por otra parte, cada uno de los sistemas operativos forman todas las operaciones de Turing a excepción de una, pudiendo dotar al atacante de suficiente poder. Por lo que en este caso, la cantidad de equipos vulnerables asciende a un 78 %. Además, se ha descubierto que la DLL *shell32* es la mejor opción en la que basar los ataques ROP en Windows gracias a que puede formar todos los conjuntos Turing-completos.

Todos estos resultados fortalecen el hecho de que los ataques ROP son una vulnerabilidad actual que afecta a una gran cantidad de equipos, animando al diseño de nuevas defensas para su mitigación.

Como líneas de trabajo futuro se plantean diversas actuaciones, como la supresión automáticas de efectos colaterales de las operaciones, extensión a otras arquitecturas,

definición de patrones de *shellcodes* usando la sintaxis definida e integración con la herramienta, y la propuesta de nuevas defensas.

Bibliografía

[Ale96] Aleph One. Smashing The Stack For Fun And Profit. *Phrack Magazine*, 7(49):File 14, 1996. `http://phrack.org/issues/49/14.html`.

[BRSS08] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 27–38. ACM, 2008.

[CDD+10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrenko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 559–572. ACM, 2010.

[CFK+09] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage. In David Jefferson, Joseph Lorenzo Hall, and Tal Moran, editors, *EVT/WOTE*. USENIX Association, 2009.

[Coma] Capstone Community. Capstone: Python tutorial for Capstone. `http://www.capstone-engine.org/lang_python.html`. Accessed: 2016-09-16.

[Comb] Capstone Community. Capstone: The Ultimate Disassembler. `http://www.capstone-engine.org`. Accessed: 2016-09-03.

[Comc] Radare2 Community. Radare2. `http://rada.re/r/`. Accessed: 2016-09-05.

[Cor] Microsoft Corporation. Calling conventions, MSDN. `https://msdn.microsoft.com/en-us/library/k2b2ssfy.aspx`. Accessed: 2016-09-08.

[Cor16] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, April 2016.

[CXS+09] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting Return-Oriented Programming Malicious Code. In Atul

Prakash and Indranil Gupta, editors, *International Conference on Information Systems Security*, volume 5905 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2009.

[Des97] Solar Designer. "return-to-libc" attack. *Bugtraq, Aug*, 1997.

[DSW09] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In Shouhuai Xu, N. Asokan, Cristina Nita-Rotaru, and Jean-Pierre Seifert, editors, *Society for Technical Communication*, pages 49–54. ACM, 2009.

[FC08] Aurélien Francillon and Claude Castelluccia. Code Injection Attacks on Harvard-architecture Devices. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 15–26, New York, NY, USA, 2008. ACM.

[Fre] Miller Freeman. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. `https://msdn.microsoft.com/en-us/library/ms809762.aspx`. Accessed: 2016-09-05.

[HHF09] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In Fabian Monrose, editor, *USENIX Security Symposium*, pages 383–398. USENIX Association, 2009.

[HSL+12] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming. In Elie Bursztein and Thomas Dullien, editors, *USENIX Workshop on Offensive Technologies*, pages 64–76. USENIX Association, 2012.

[Kor09] Tim Kornau. Return oriented programming for the ARM architecture. Master's thesis, Ruhr-Universität Bochum, Germany, 2009. `https://static.googleusercontent.com/media/www.zynamics.com/es//downloads/kornau-tim--diplomarbeit--rop.pdf`.

[Lin09] Felix Lindner. Developments in Cisco IOS forensics. CONFidence 2.0, 2009.

[MP88] Farhad Mavaddat and Behrooz Parhami. URISC: the ultimate reduced instruction set computer. *International Journal of Electrical Engineering Education*, 25(4):327–334, 1988.

[RBSS12] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, 15(1):2:1–2:34, March 2012.

[Sal] Jonathan Salwan. ROPgadget Tool. `https://github.com/JonathanSalwan/ROPgadget`. Accessed: 2016-09-03.

[Sch] Sascha Schirra. Ropper. `https://github.com/sashs/Ropper`. Accessed: 2016-09-03.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM Conference on Computer and Communications Security*, pages 552–561. ACM, 2007.

[SPP+04] Shacham, Page, Pfaff, Goh, Modadugu, and Boneh. On the Effectiveness of Address-Space Randomization. In *SIGSAC: 11th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2004.

[Tea03] PaX Team. PaX non-executable pages design & implementation. *Available: http://pax.grsecurity.net*, 2003.

[Tur36] Alan Turing. On Computable Numbers with an Application to the Entscheidungs Problem. *Proc. London Mathematical Society*, 2(42):230–265, 1936.

[W3S] W3Schools. OS Platform Statistics. `http://www.w3schools.com/browsers/browsers_os.asp`. Accessed: 2016-09-09.

Apéndice A

Horas de trabajo

El desarrollo del trabajo se ha prolongado durante 7 meses, controlando en este periodo el tiempo empleado en cada una de las tareas. En la Figura A.1 se muestra un Diagrama de Gantt que representa el esfuerzo dedicado desglosado por semanas y tareas.

El trabajo ha consistido en diferentes tareas: (i) estudiar los ataques ROP, su evolución y las defensas para contenerlos; (ii) realizar una comparación de los diferentes *frameworks* disponibles para el desensamblado de binarios; (iii) estudiar en detalle de la API que ofrece Capstone Disassembly Framework para Python; (iv) estudiar la máquina de Turing y especificar sus operaciones mediante conjuntos de *gadgets*; (v) analizar, diseñar e implementar la herramienta para encontrar las operaciones de Turing mediante los conjuntos definidos; y finalmente, todo este proceso ha permitido (vi) realizar una experimentación y (vii) su posterior recopilación de resultados.

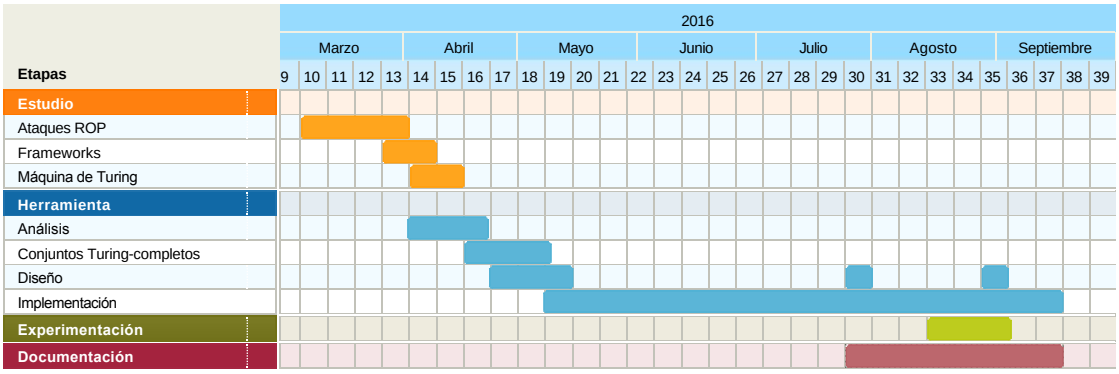


Figura A.1: Diagrama de Gantt con el esfuerzo invertido por semanas.

En total, el trabajo ha contado con una inversión de **420 horas**. Para complementar la planificación del proyecto, se muestra en la Figura A.2 una visión de la cantidad de trabajo dedicada a cada una de las tareas.

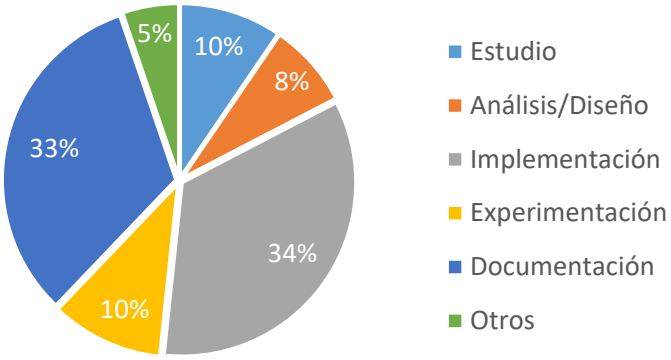


Figura A.2: Esfuerzo dedicado por tareas.

Apéndice B

Catálogo de conjuntos

Cargar una constante (lc)

```
pop dst;
```

```
popad;
```

Mover un valor (move)

```
xor dst, dst;
add dst, srx;

and dst, src; (dst = 0xFFFFFFFF)

xor dst, dst;
not dst;
and dst, src;

xor dst, dst;
stc;
adc dst, src;

xor aux, aux;
xor dst, dst;
neg aux;
adc dst, src;

clc;
cmovnc dst, src;
```

```
xor aux, aux;
neg aux;
cmovc dst, src;

neg aux (aux = 0xFFFFFFFF)
cmovc dst, src;

stc;
cmovc dst, src;

mov dst, src;

push src;
pop dst;

xchg dst, src;

xor eax, eax;
inc eax;
mul dst;
```

```
xor src, src;
xadd src, dst;
```

```
xor dst, dst;
neg src;
sub dst, src;
```

Cargar un valor (load)

```
mov dst, [src]
```

Guardar un valor (store)

```
mov [dst], src
```

Sumar (add)

```
add dst, src;
```

```
clc;
adc dst, src;
```

```
xor aux, aux;
neg aux;
adc dst src;
```

Restar (sub)

```
sub dst, src;
```

```
clc;
sbb dst, src;
```

```
xor aux, aux;
neg aux;
sbb dst, src;
```

And (and)

```
and dst, src;
```

Or (or)

```
or dst, src;
```

Xor (xor)

```
xor dst, src;
```

Not (not)

```
not dst;
```

```
xor dst, src; (dst = 0xFFFFFFFF)
```

Salto condicional (cond1)

```
sub dst, src;
xor aux, aux;
adc aux, aux;
neg aux;
```

Salto condicional (cond2)

```
pop dst;
and src, dst;
add esp, src
```

```
pop dst;
and src, dst;
add src, esp;
mov esp, src;
```

```
pop dst;
and src, dst;
add src, esp;
xchg esp, src;
```