Return-Oriented Programming without Returns

Seguridad Informática

Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

Agosto 2025

Inyección de Código

Una forma de tomar el control de un proceso es: aprovechar un desbordamiento de arreglo en la pila.
Pisar la dirección de retorno y saltar a una parte del código o ejecutar código en la misma pila.

Inyección de Código

Una forma de tomar el control de un proceso es: aprovechar un desbordamiento de arreglo en la pila.
Pisar la dirección de retorno y saltar a una parte del código o ejecutar código en la misma pila.
Contramedidas:

- Programación fuertemente tipada y validar longitudes.
- No utilizar funciones de biblioteca y diseñar de forma segura.
- Minimizar los puntos de error, capacitar al personal (y tenerlo).
- Verificar los retornos.
- Colocar la pila como no ejecutable y agregar canarios.

Inyección de Código

Una forma de tomar el control de un proceso es: aprovechar un desbordamiento de arreglo en la pila.
Pisar la dirección de retorno y saltar a una parte del código o ejecutar código en la misma pila.
Contramedidas:

- Programación fuertemente tipada y validar longitudes.
- No utilizar funciones de biblioteca y diseñar de forma segura.
- Minimizar los puntos de error, capacitar al personal (y tenerlo).
- Verificar los retornos.
- Colocar la pila como no ejecutable y agregar canarios.

Programación orientada a los retornos[1]

Permite explotar errores de memoria sin inyectar código. Aprovechando la instrucción `return` pues realizan un salto y cambia el estado del procesador.

[1] *Return-Oriented Programing*

Programación orientada a los retornos[1]

Permite explotar errores de memoria sin inyectar código.
Aprovechando la instrucción `return` pues realizan un salto y cambia el estado del procesador.
Contramedidas:

- Revisar los flujos de instrucciones con retornos frecuentes.
- Validar el invariante de *last-in*, *first-out*.
- No usar la instrucción `return`.

[1] *Return-Oriented Programing*

conjunto de *Gadget*

Definición

Secuencia de instrucciones que ya están presentes en el proceso, en particular tienen permisos de ejecución. El stack es usado como lugar para almacenar el "código".

En la programación orientada a los retornos suelen finalizar con la instrucción `return`.

Permiten definir una máquina virtual.

conjunto de *Gadget*

Definición

Secuencia de instrucciones que ya están presentes en el proceso, en particular tienen permisos de ejecución. El stack es usado como lugar para almacenar el "código".

En la programación orientada a los retornos suelen finalizar con la instrucción `return`.

Permiten definir una máquina virtual.

Se puede buscar posibles *Gadget* mediante algoritmos.

Convención de llamada

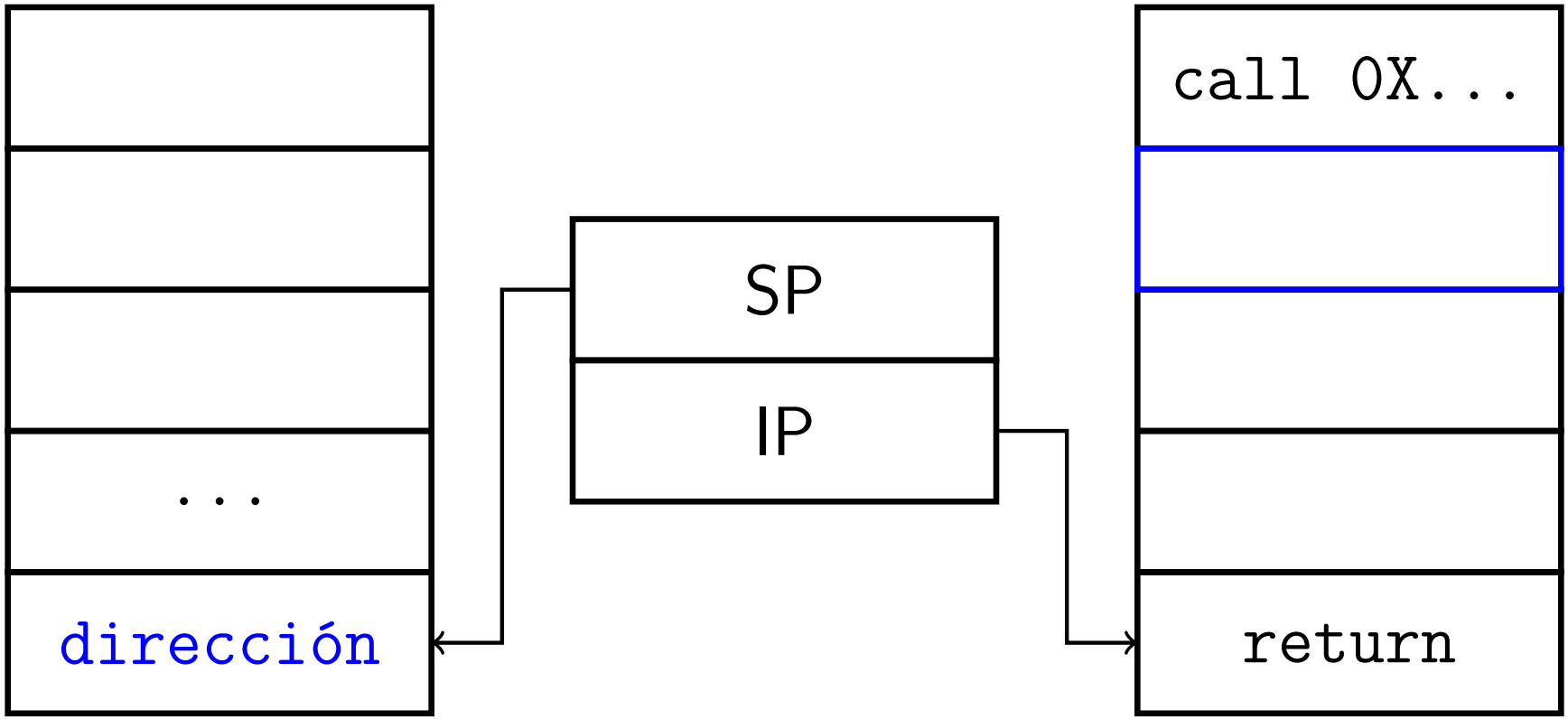


Figure: Estado de los registro para la pila e instrucciones antes de un `return`.

Convención de llamada

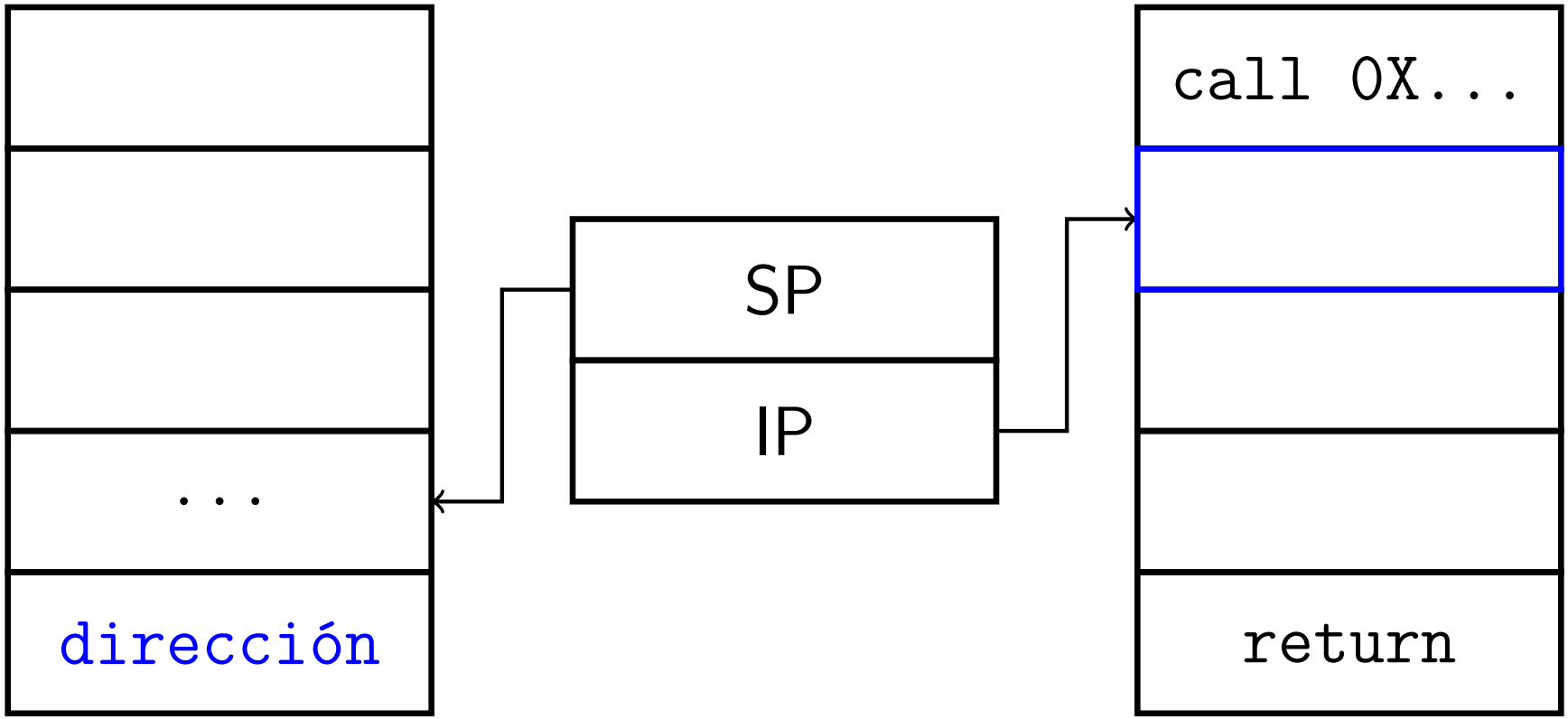


Figure: Estado de los registro para la pila e instrucciones después de un `return`.

Programación orientada a los retornos sin retornos[2]

Se asume:

- Puede haber implementado un modelo de lectura o ejecución exclusivos ($W \oplus X$).
- Puede haber contramedidas para la programación orientada a los retornos.
- Hay una aplicación vulnerable a los ataques.

Necesitamos otras formas de tener las 2 propiedades del `return`:

1. Trasferir el control de ejecución con un salto indirecto.
2. Actualizar el estado del proceso (no volver a la misma instrucción)

[2] *Return-Oriented Programing without Returns*

Si no usamos return

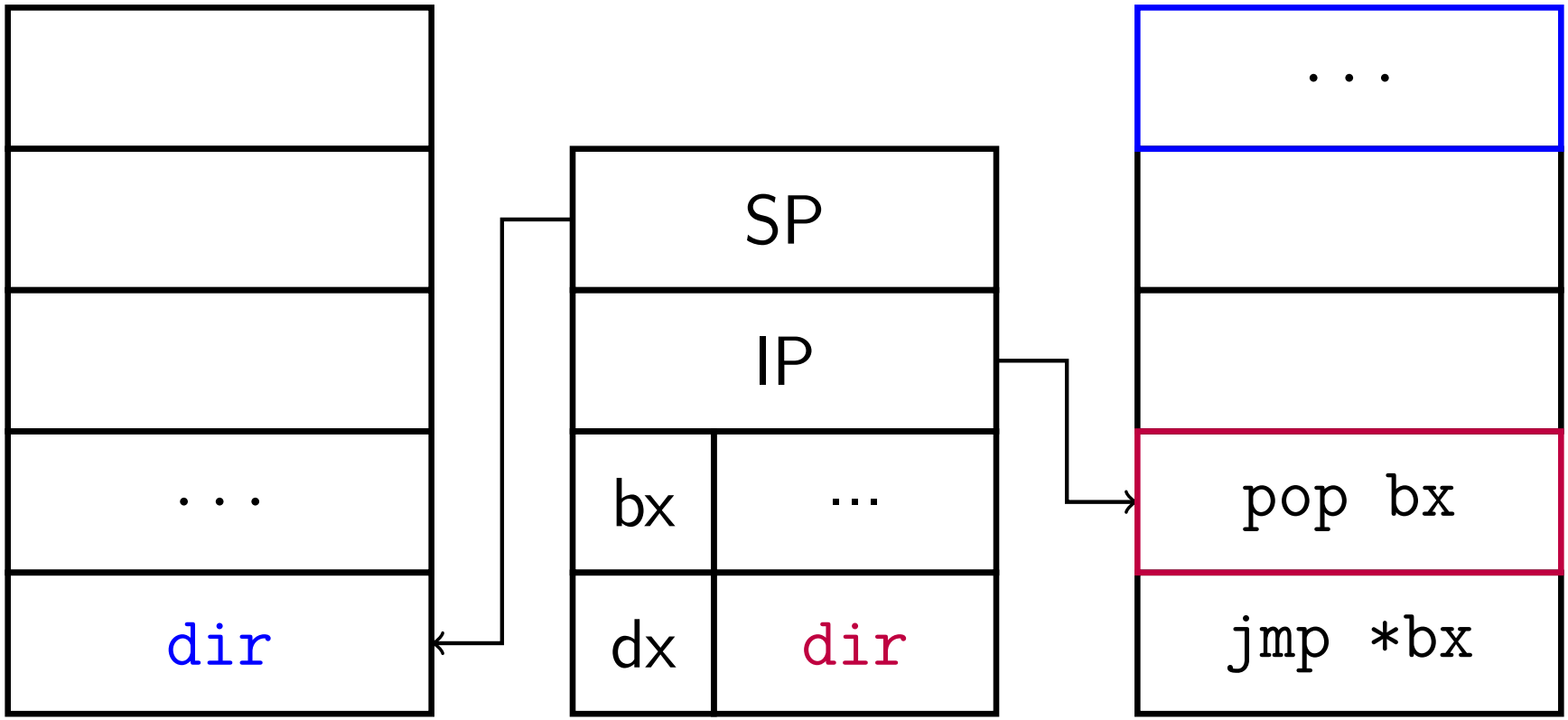


Figure: Ejecución del "trampolín": pop saca de la pila y coloca en **bx**

Si no usamos return

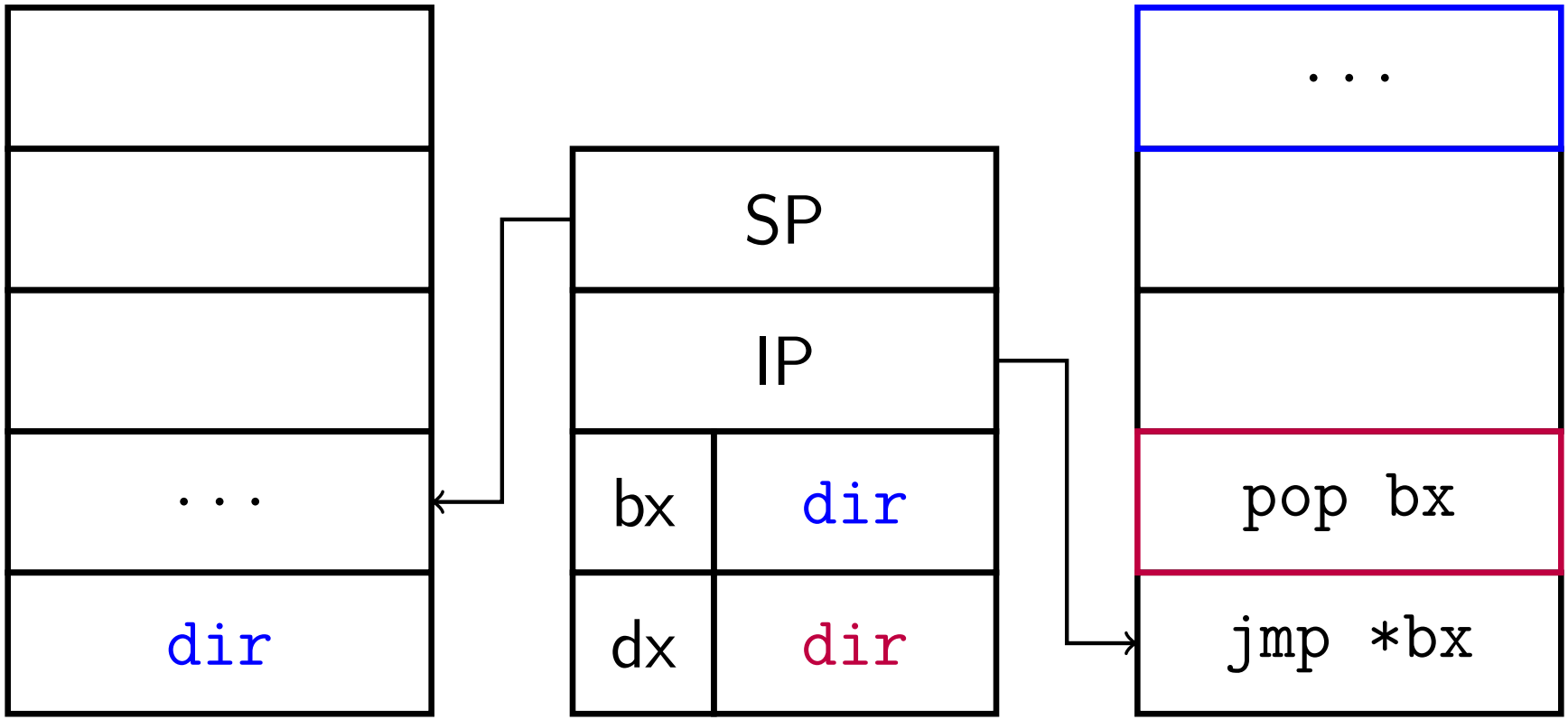


Figure: Ejecución del "trampolín": `jmp` salta a la dirección de **bx**

Si no usamos return

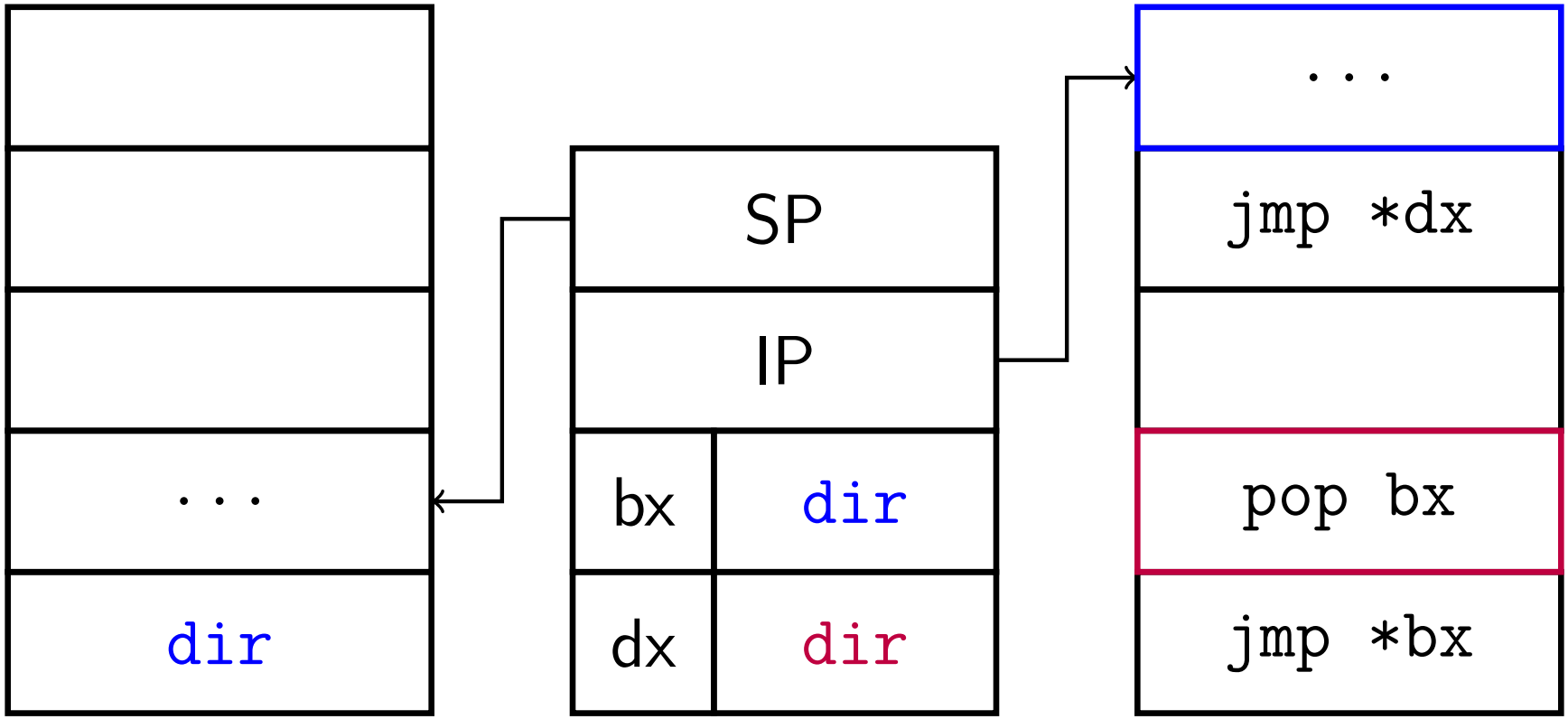


Figure: Ejecución del código del *gadget*. Debe terminar con un salto al trampolín.

INTEL x86

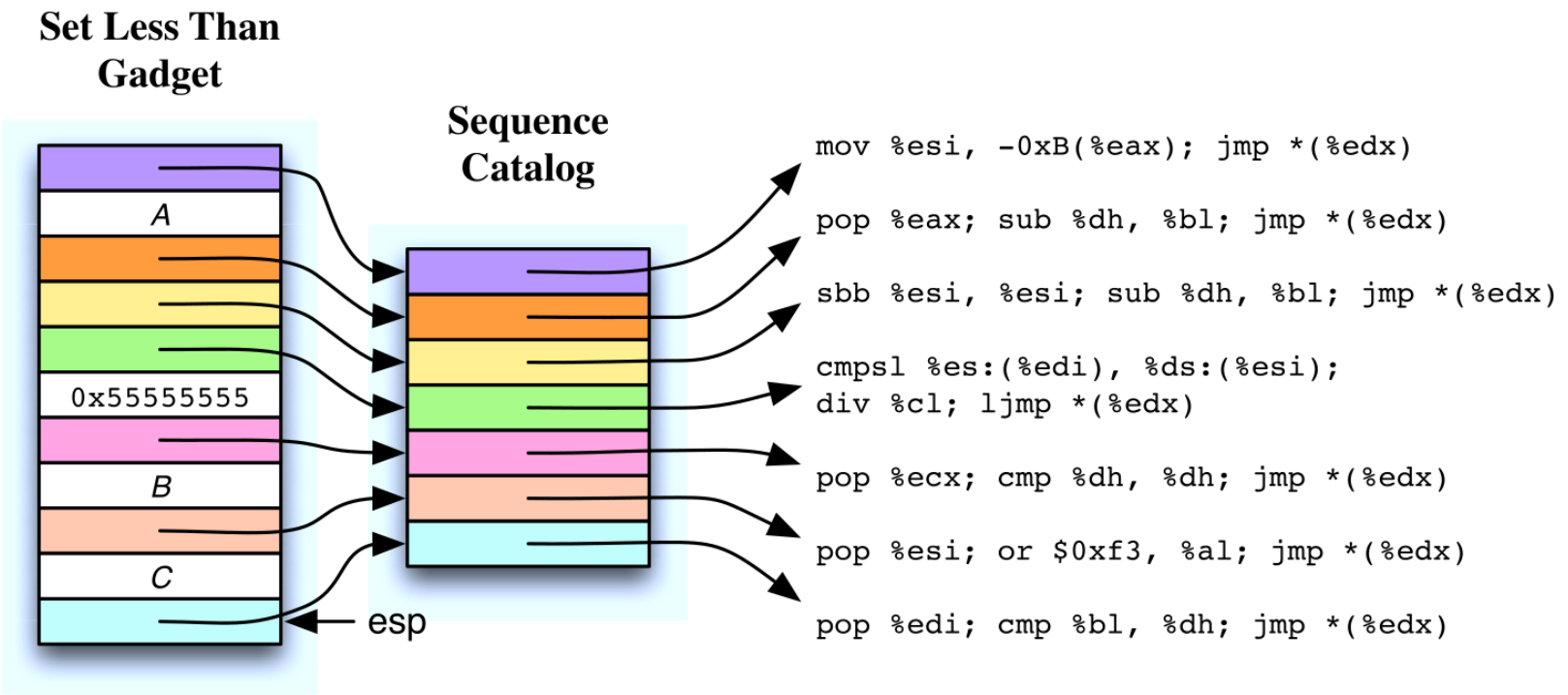
En este caso el conjunto de *gadgets* proviene de **libc** y **libxul**.
Que el compilador evite su aparición es complicado.

Los *gadgets* son 19: `load (immediate)`, `move`, `store`, `add (immediate)`, `subtract`, `negate`, `and (immediate)`, `or (immediate)`, `xor (immediate)`, *complement*, `branch unconditional`, `branch conditional`, `set less than` y `call`.

INTEL x86 (observaciones)

- Se busca que las operaciones actualicen valores en memoria.
- Las interacciones con memoria requieren registros intermedios, porque se usan 2 niveles de indirección.
- Las ramas condicionales actualizan valores de memoria y no el registro de instrucciones.
- La convención para hacer llamadas a funciones utilizada en el paper no es la actual en Linux.

INTEL x86 (*gadget*)

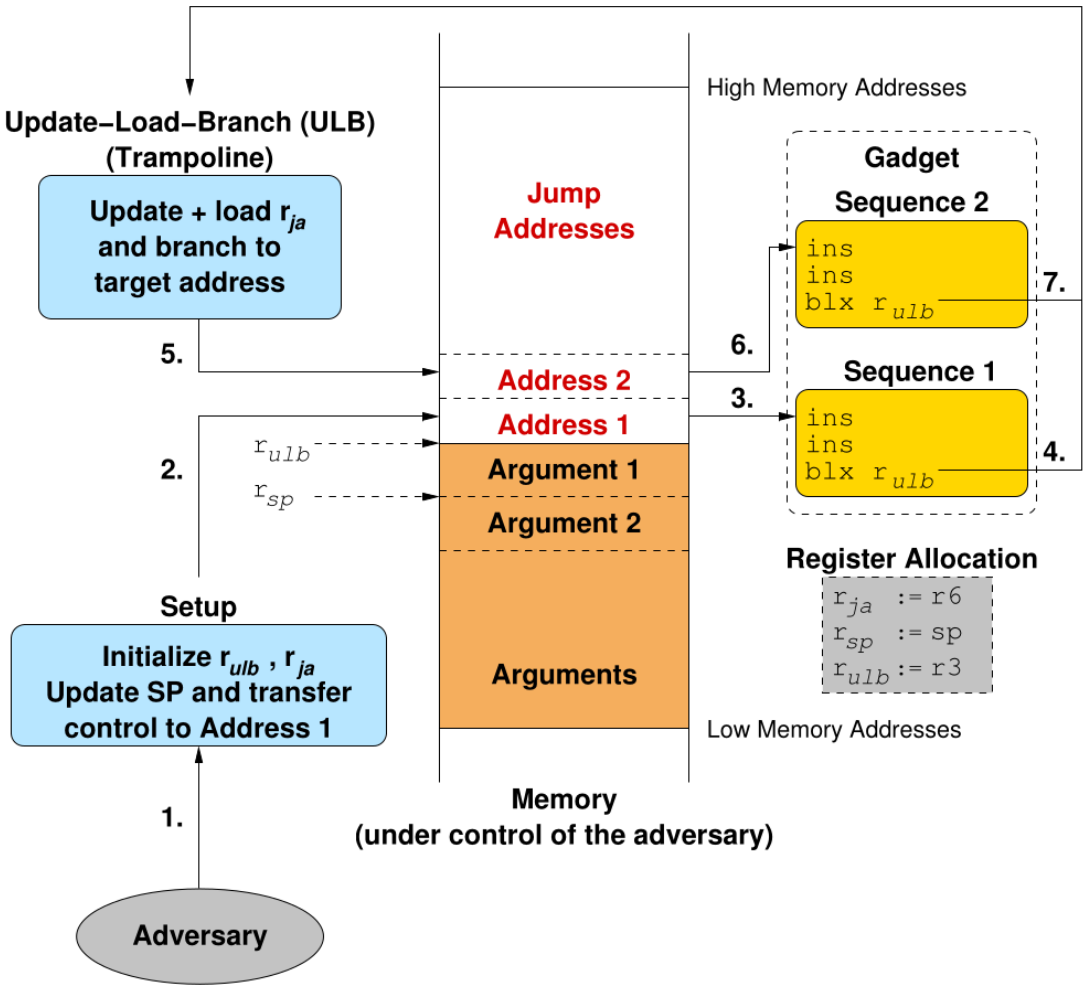


ARM

En ARM no tenemos algo como la convención de llamada en x86. El epílogo no funciona igual, hay 16 registros los cuales se pueden modificar directamente (incluido el de instrucciones).

La instrucción `bl` y `blx` son la que nos permiten los saltos pero solo `blx` permite saltar con direcciones en registros. Desde donde se salta se almacena en r14 (lr).

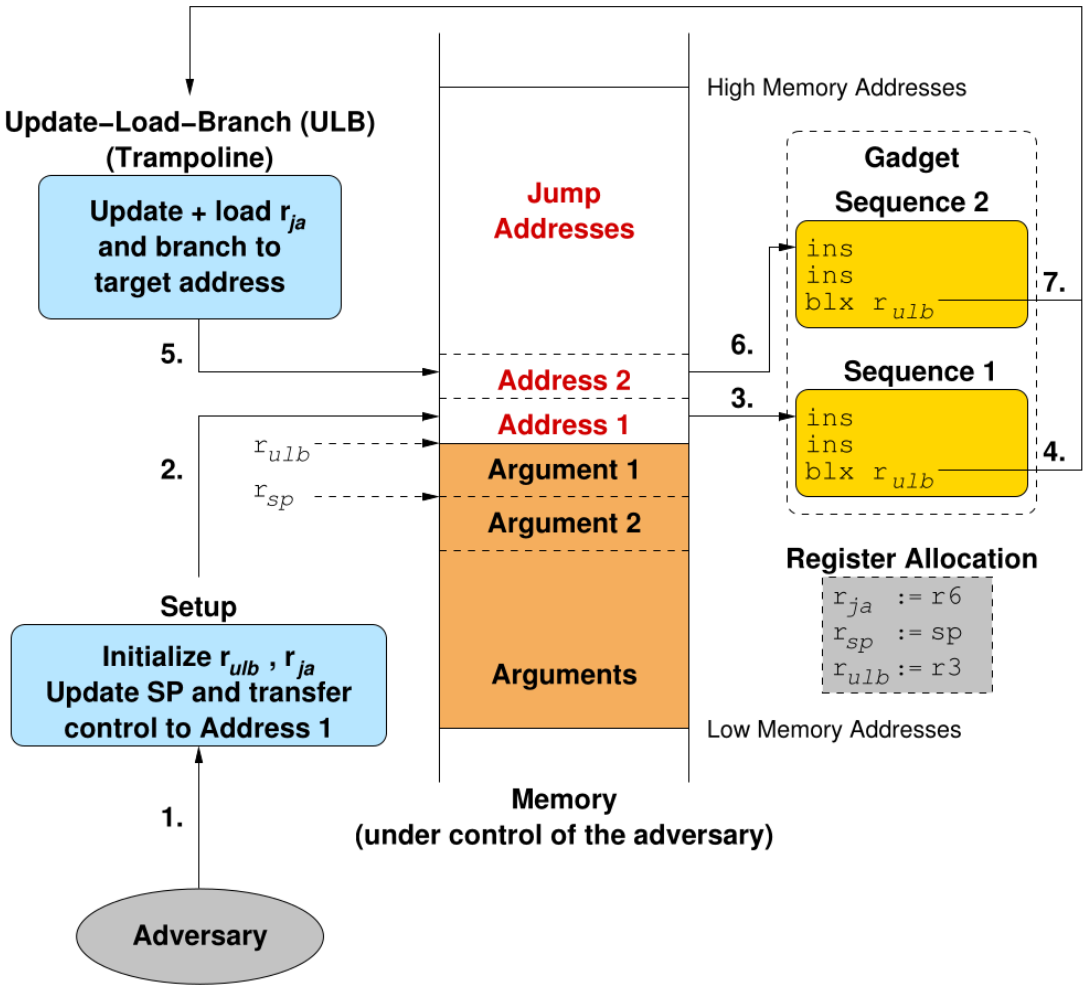
ARM



ULB

```
adds r6, #4;
ldr r5, [r6, #124];
blx r5;
```

ARM



Update Load Branch

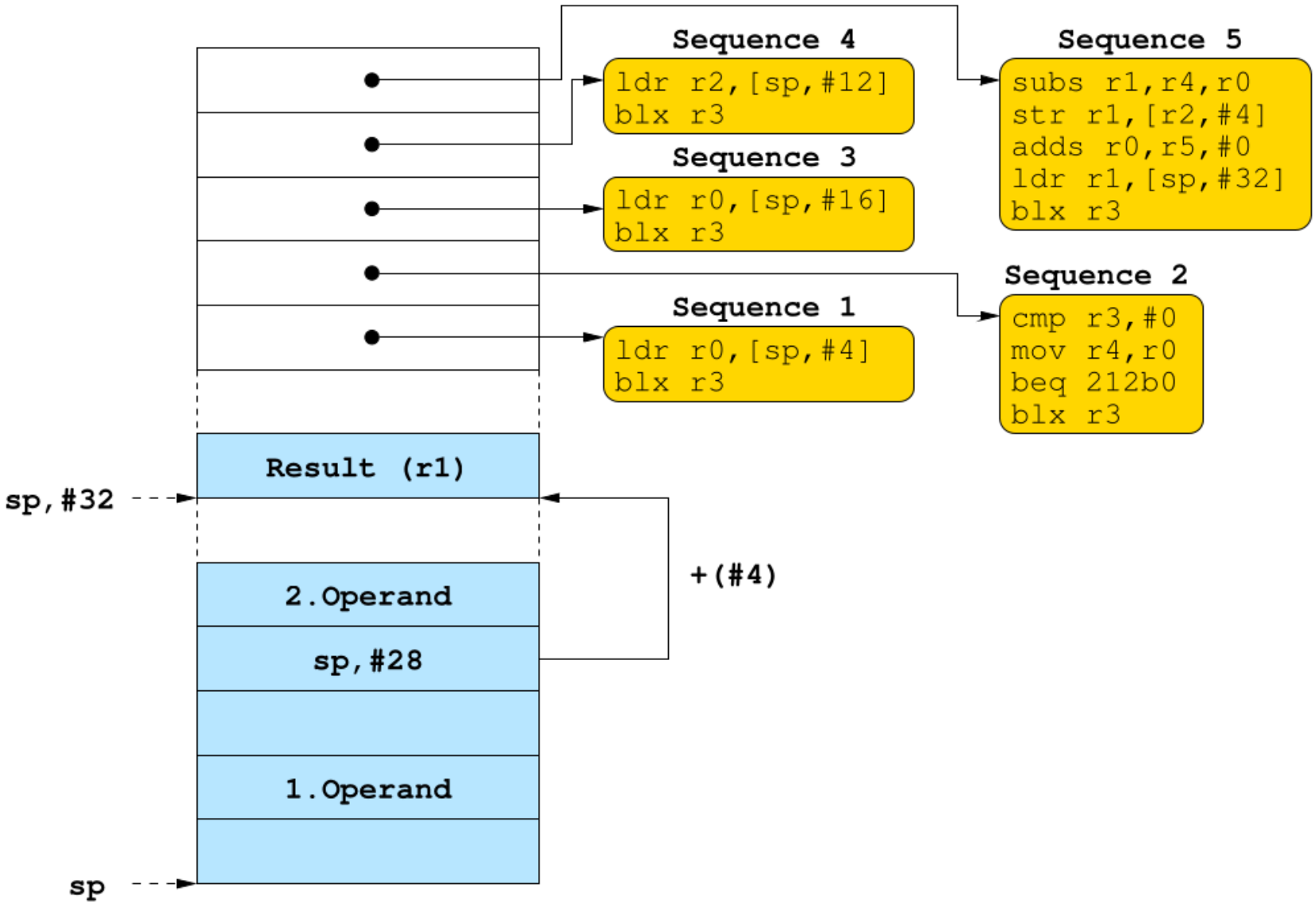
```
adds r6, #4;
ldr r5, [r6, #124];
blx r5;
```

ARM (observaciones)

- Las instrucciones deben terminar con `blx`.
- Se construyeron los *gargets* con **libc** y **libwebcore**
- Se requieren 3 registros en lugar de 1.
- Por limitaciones de la arquitectura hacen que manejar memoria sea más complejo.
- La forma de realizar operaciones aritmético-lógicas es idéntica, igualmente los condicionales.
- Este ataque no puede ser detectado por *validadores de direcciones de retorno*[3]. No se modifica la pila.

[3] return address checker

ARM (*gadget*)


Sequence 4
ldr r2,[sp,#12]
blx r3
Sequence 5
subs r1,r4,r0
str r1,[r2,#4]
adds r0,r5,#0
ldr r1,[sp,#32]
blx r3
Sequence 3
ldr r0,[sp,#16]
blx r3
Sequence 2
cmp r3,#0
mov r4,r0
beq 212b0
blx r3
Sequence 1
ldr r0,[sp,#4]
blx r3
Result (r1)
sp,#32
+(#4)
2.Operand
sp,#28
1.Operand
sp

Linux Intel x86

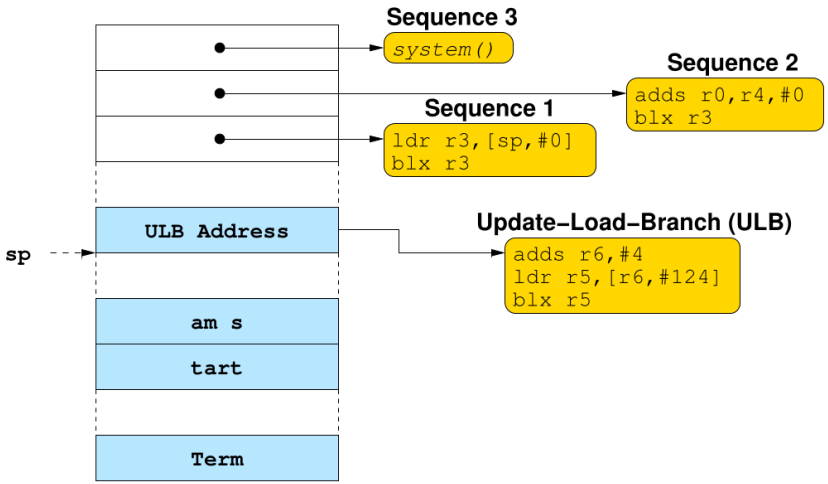
```
struct foo {
    char buffer[160];
    jmp_buf jb;
};

int main( int argc, char **argv ) {
    struct foo *f = malloc( sizeof *f );
    if( setjmp(f->jb) )
        return 0;
    strcpy( f->buffer, argv[1] );
    longjmp( f->jb, 1 );
}
```

Se dá un *shellcode egg* a este programa con:

- El programa (ROPWR).
- Los datos (programa a ejecutar).
- el catálogo de instrucciones (doble indirección).
- Datos para pisar `foo`.

Google Android ARM



Obtener una shell en el emulador para el desarrollador de android.
Se utiliza el mismo código que antes a traves de una interfaz de java para C.

Conclusiones

- Encontrar un trampolín basta para construir a su alrededor un conjunto de *gargets*.
- Esta nueva forma de programar es un problema grave de seguridad.
- Las posibles mitigaciones pueden tener una gran penalidad.