Universidad Nacional de Córdoba
Facultad de Matemáticas, Astronomía y Física

Verificación Formal de Código Binario

David Daniel Arch

Directores
Dr. Damian Barsotti
Dr. Pedro R. D'Argenio

Esta página queda intencionalmente en blanco.

Resumen

Los buffer overflow son una de las vulnerabilidades mas frecuentes que se encuentran en el software que utilizamos diariamente. El análisis de este tipo de vulnerabilidades se hace tedioso y complejo cuando solo se tiene acceso al código binario y no al código fuente del programa vulnerable debido al uso del lenguaje de bajo nivel ensamblador, la poca estructura que presenta este lenguaje, la ausencia de funciones bien definidas y de tipos de datos abstractos entre otros. Con el objetivo de facilitar el trabajo de los investigadores de seguridad presentamos una extensión para el framework BAP que agrega soporte para la verificación de buffer overflow en código binario mediante el uso de SMT solvers.

Palabras clave: BAP, SMT, Verificación, Codigo Binario, Buffer Overflow
Clasificación: D.2.4 Software/Program Verification [Formal Methods]

Esta página queda intencionalmente en blanco.

Índice general

Capítulo 1

Introducción

Se denomina malware o software malicioso a cualquier software que tiene como objetivo alterar el funcionamiento normal, robar datos sensitivos, o acceder de manera no autorizada a un sistema. Con el pasar de los años la cantidad de malware existente ha crecido exponencialmente. En los últimos años esta tendencia parece haber cesado, no porque los desarrolladores estén produciendo software más seguro, sino porque el malware que esta siendo desarrollado es cada vez más sofisticado, tanto en las técnicas de explotación de vulnerabilidades, como en las técnicas de ofuscación, lo que supone un mayor tiempo de desarrollo. Un ataque ocasionado por este tipo de programas en un ámbito corporativo puede suponer una gran pérdida económica. Es por esto que un análisis rápido del código binario malicioso es indispensable para reducir las pérdidas potenciales. Generalmente, los investigadores de seguridad no tienen acceso al código fuente de los archivos maliciosos, por lo que es necesario trabajar directamente sobre código binario para poder determinar el funcionamiento del malware y de esta forma lograr mitigar los efectos en el sistema. Trabajar con código binario es, generalmente, un proceso lento y tedioso, ya que no existen abstracciones como las que proveen los lenguajes de programación de alto nivel, tipos, estructuras de datos o funciones bien definidas, a esto se suma la gran cantidad de instrucciones existentes y la falta de una semántica bien definida para razonar sobre los programas. Frameworks como BAP[15] tratan este último problema mediante el uso de un lenguaje intermedio que hace explícita la semántica de cada una de las instrucciones de ensamblador.

Descripción del trabajo

En este trabajo desarrollaremos un método de análisis estático para detectar la presencia de buffer overflow en código binario utilizando el lenguaje de programación Python. El método desarrollado toma como punto de partida el framework de análisis binario BAP, tanto para convertir el código binario en un lenguaje intermedio sobre el que se realizará el análisis, como para realizar la verificación del mismo utilizando métodos formales.

Organización de la tesis

En el capítulo 2 veremos el marco teórico necesario para la comprensión de la tesis. En el capítulo 3 describiremos el método desarrollado que nos permitirá detectar la presencia de buffer overflow en binarios utilizando BAP. En el capítulo 4 veremos algunos casos de uso en los que se aplicó la técnica desarrollada. En el capítulo 5 veremos como podría ser mejorado y extendido este trabajo. Finalmente, en el capítulo 6, presentaremos las conclusiones.

Capítulo 2

Marco Teórico

Este capítulo esta organizado de la siguiente manera: en la sección 1 veremos algunas nociones básicas del lenguaje ensamblador y la semántica de algunas instrucciones comunes que veremos a lo largo de la tesis. En la sección 3 explicaremos el funcionamiento del stack, sus usos en la programación en ensamblador y la definición de buffer overflow. En la sección 4 veremos el problema de la satisfacibilidad proposicional y algunas de las teorías de SMT que usaremos en el método desarrollado. La sección 5 está dedicada a BAP, su funcionamiento, el lenguaje intermedio y las funcionalidades más relevantes que utilizaremos. En la sección 6 desarrollaremos el lenguaje JSON y como se representa el lenguaje intermedio de BAP en este formato.

2.1. Ensamblador

El ensamblador es un lenguaje de programación de bajo nivel, relacionado directamente a las instrucciones del hardware de una computadora, por lo tanto el lenguaje ensamblador de cada arquitectura de computadora tiene su propio juego de instrucciones. Este juego de instrucciones no es compatible entre distintas arquitecturas. Algunos ejemplos de arquitecturas de computadora son x86, x86_64, MIPS, ARM, etc. Nosotros trabajaremos sobre la arquitectura x86 de 32 bits.

2.1.1. Arquitectura x86

La arquitectura x86 esta conformada por una serie de procesadores que salieron al mercado posteriores al Intel 8086. Si bien los primeros procesadores de esta familia funcionaban con 16 bits, los procesadores a partir del 80386 y hasta la llegada de x86-64 funcionan con 32 bits. Cuando utilicemos este

término estaremos haciendo referencia a procesadores modernos de 32 bits.

La unidad más pequeña de datos con la que trabajan estos procesadores es el byte, cada dirección de memoria hace referencia a un conjunto de 8 bits, siendo imposible referenciar una cantidad menor de bits.

Estos procesadores generalmente tiene un bus de direcciones de 32 bits, lo que permite direccionar 4294967296 direcciones únicas de memoria (2ˆ32). Es decir que la cantidad máxima de memoria que podemos direccionar, dado que cada dirección puede contener 1 byte de datos, es de 4 GB.

2.1.1.1. Registros

Los registros son la unidad de almacenamiento de datos más pequeña y rápida de la que dispone la CPU para operar.

La arquitectura x86 posee 8 registros de uso general (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP), esto quiere decir que pueden ser utilizados por el programador libremente. Sin embargo, el uso de estos registros está optimizado para determinadas funciones.

Explicaremos ahora, brevemente, las funcionalidades de estos registros.

El registro EAX es usado para realizar cálculos, por ejemplo, para operaciones aritméticas, y para almacenar el valor de retorno de las funciones.

El registro EDX se utiliza como auxiliar del registro EAX para operaciones que requieran del almacenamiento de datos extra, por ejemplo para almacenar el resto de la división y también para realizar operaciones de entrada y salida.

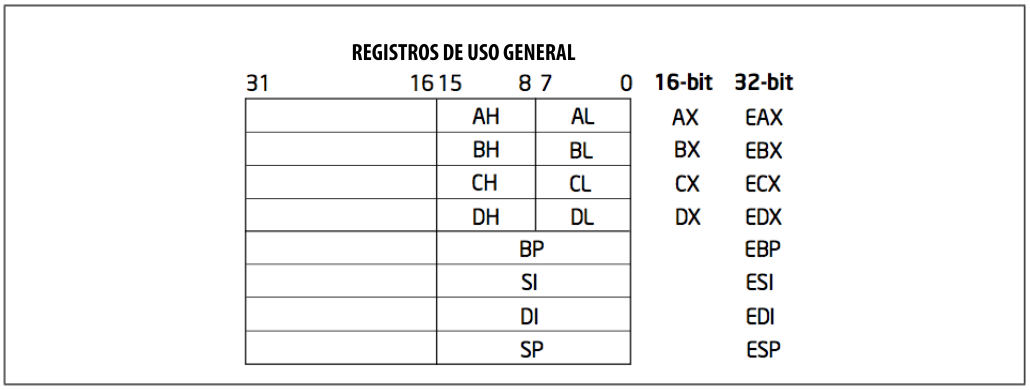


Figura 2.1: Registros de Propósito General x86

El registro ECX se utiliza como contador en las operaciones de ciclos y en las operaciones de copiado de strings.

Los registros ESI y EDI, se utilizan para almacenar las direcciones de origen y destino respectivamente que serán utilizadas en operaciones de manipulado de datos, por ejemplo, el copiado de strings.

Los registros ESP y EBP se utilizan en las operaciones relacionadas con el stack. El registro ESP guarda la dirección de memoria del tope de la pila, mientras que EBP se utiliza como base del stack frame actual. El funcionamiento de estos registros se explicará con más detalle en la próxima sección.

El registro EBX se utiliza para almacenar punteros a datos.

El registro EIP (program counter) guarda la dirección de la próxima instrucción a ejecutar por la CPU. Hay que aclarar que no existen instrucciones que nos permitan acceder o controlar directamente al valor de este registro. Las únicas instrucciones que pueden modificar indirectamente su valor son las instrucciones de control de flujo que veremos más adelante.

Además de estos registros tenemos otros registros que nos permiten acceder a los 16 bits menos significativos de cada uno de los registros nombrados anteriormente, estos son AX, BX, CX, DX, BP, SI, DI, SP. A su vez, para AX, BX, CX y DX existen registros que nos permitirán acceder a los 8 bits más significativos y 8 bits menos significativos de cada registro de 16 bits. En el caso de AX, con el registro AL accederemos a los 8 bits menos significativos y con AH a los más significativos. En la Figura 2.1 podemos apreciar esto gráficamente.

Además de estos registros, existe un registro especial de denominado EFLAGS. Este registro, también de 32 bits, esta compuesto de 32 valores booleanos.

Los flags se pueden clasificar, como podemos ver en la Figura 2.2 en:

- Flags de estado
- Flags de control
- Flags de sistema

Los flags de estado se utilizan para indicar el resultado de operaciones aritméticas como ADD, SUB, etc. Estos son:

- Carry flag: se setea cuando una operación aritmética produce acarreo o cuando se produce un desbordamiento en una operación aritmética sin signo.
- Parity flag: se setea si el byte menos significativo del resultado tiene una cantidad par de bits.
- Auxiliar Carry flag: este flag se utiliza para aritmética BCD[1].

[1] https://es.wikipedia.org/wiki/Decimal_codificado_en_binario

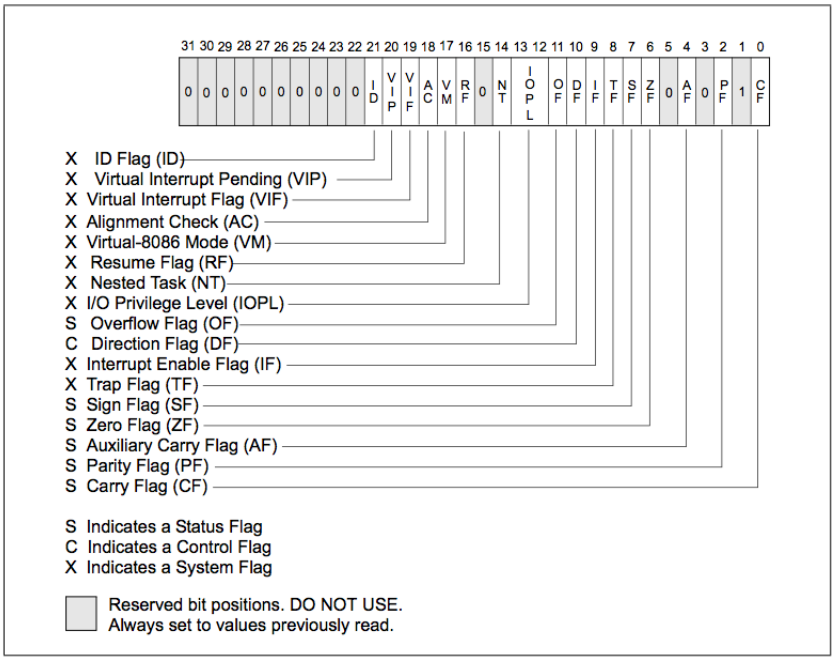


Figura 2.2: Registro de EFLAGS.

- Zero flag: se setea cuando el resultado de una operación es cero.
- Sign flag: su valor es igual al bit más significativo del resultado de una operación (bit de signo de un entero).
- Overflow flag: se setea cuando una operación produce un resultado demasiado grande para ser almacenado en el destino.

El único flag de control es el direction flag, este determina si las operaciones de copiado de memoria se realizarán hacia direcciones superiores o inferiores de memoria. En general no existen instrucciones que nos permitan modificar directamente el estado de los flags, con la excepción de los flags de dirección (DF) y carry flag (CF). Las instrucciones que nos permiten modificar el flag de dirección son STD y CLD, que encienden y apagan el flag respectivamente. Con el flag prendido, el copiado de memoria se realiza desde direcciones altas de memorias a direcciones más bajas de memoria, con el flag apagado es exactamente al revés.

Finalmente, los flags de sistema, marcados con una X en la Figura 2.2 son utilizados por el sistema operativo y no deben ser modificados por el programador.

2.1.1.2. Instrucciones

El juego de instrucciones del 8086 tenía alrededor de 80 instrucciones. Con cada nueva generación de procesadores de la familia x86 se fueron agregando más y más instrucciones. El manual del desarrollador de la arquitectura x86 [20] tiene alrededor de 4000 paginas actualmente. Utilizaremos la sintaxis utilizada por Intel para escribir las instrucciones ensamblador en lo que resta de esta tesis.

Instrucciones de Transferencia de Datos Estas instrucciones nos permiten realizar el copiado de datos.

Ahora explicaremos las instrucciones que utilizaremos a lo largo de la tesis.

MOV La sintaxis de una instrucción MOV es:

```
MOV dst, src
```

src puede ser un valor inmediato, un registro de uso general o el contenido de una dirección de memoria (en este caso la sintaxis es [src]).

dst puede ser un registro de uso general o una dirección de memoria (en este caso la sintaxis es [dst]).

La semántica de MOV es colocar el valor de *src* en el destino *dst*.

Esta instrucción no afecta el registro de EFLAGS.

PUSH La sintaxis de una instrucción PUSH es:

```
PUSH src
```

src puede ser un registro o un valor directo.

En cuanto a su semántica, debemos recordar que el registro ESP contiene la dirección de memoria del tope del stack. La semántica de PUSH es decrementar el valor de ESP en 4 (se decrementa porque el stack crece hacia direcciones inferiores de memoria y en 4 ya que cada dirección de memoria puede almacenar 8 bits, y la arquitectura en la que trabajamos trabaja con 32 bits) y escribir el valor en la dirección de memoria a la que apunta ESP. El nuevo tope del stack apuntará a nuestro valor recientemente agregado.

Esta instrucción no afecta el registro EFLAGS.

POP La sintaxis de una instrucción POP es:

```
POP dst
```

dst solo puede ser un registro.

En cuanto a la semántica de la instrucción, POP coloca el valor que se encuentra en el tope del stack en el registro destino y luego incrementa ESP en 4. Es importante resaltar que esta operación no modifica el valor en memoria, solamente lo copia en el destino.

Podemos notar que el stack crece hacia direcciones de memoria más bajas a medida que vamos pusheando valores y se reduce cuando vamos popeando valores hacia direcciones de memoria más altas.

Esta instrucción no afecta el registro de EFLAGS.

Instrucciones de Control de Flujo Estas instrucciones nos permiten controlar la ejecución de un programa. Sin estas instrucciones no sería posible realizar saltos condicionales o ciclos dentro de los programas.

CALL La sintaxis de una instrucción CALL es:

```
CALL dst
```

dst puede ser un valor inmediato, un registro de uso general o una dirección de memoria (en este caso la sintaxis es [dst]).

La semántica de CALL es realizar PUSH de la dirección de la próxima instrucción que sigue a *dst,* y luego transferir la ejecución a *dst.*

Esta instrucción no afecta el registro de EFLAGS.

RET La sintaxis de una instrucción RET es:

```
RET
```

La semántica de RET es tomar el valor almacenado en la dirección de memoria apuntada por ESP, transferir la ejecución a ese valor e incrementar ESP en 4.

Esta instrucción no afecta el registro de EFLAGS.

Instrucciones Aritméticas Estas instrucciones nos permiten realizar sumas, restas, multiplicaciones y divisiones entre los distintos tipos de datos.

SUB La sintaxis de una instrucción SUB es:

```
SUB dst, src
```

src puede ser un valor inmediato, un registro de uso general o el contenido de una dirección de memoria (en este caso la sintaxis es [src]).

dst puede ser un registro de uso general o una dirección de memoria (en este caso la sintaxis es [dst]).

La semántica de SUB es restar *src* a *dst*, el resultado de la operación queda guardado en *dst*.

Esta instrucción modifica los siguientes flags: OF, SF, ZF, AF, PF y CF.

Instrucciones Lógicas Estas instrucciones nos permiten realizar operaciones lógicas entre los datos como conjunción, disyunción, negación, etc.

Algunos ejemplos son: *AND, OR, NEG, XOR,* etc.

Instrucciones de Shift y Rotate Estas instrucciones nos permiten realizar rotaciones y desplazamientos de los datos.

Algunos ejemplos son: *SHR, SAL, SAR,* etc.

2.2. Stack

El stack es una zona de la memoria en la ejecución de cada programa que se utiliza principalmente para:

- Guardar las variables locales a una función.
- Llamado de funciones(se explicará en 2.2.2).

Esta zona de memoria es particularmente importante para este trabajo ya que nos concentraremos en detectar buffer overflow en variables que se encuentren en esta región.

Entender el funcionamiento del stack será crucial para poder saber como se producen los buffer overflow.

2.2.1. Funcionamiento

Las dos operaciones fundamentales que se pueden realizar en el stack son:

- Agregar datos
- Remover datos

El stack funciona como una LIFO, es decir que el último elemento que hayamos agregado será el primero en ser removido. Las dos instrucciones que nos permiten trabajar sobre el stack son *PUSH* y *POP* (sección 2.1.1.2) para agregar y remover datos respectivamente.

Una cosa a tener en cuenta es que el stack crece hacia direcciones de memoria decrecientes, esto se debe a la semántica de PUSH. Cada vez que agregamos datos en el stack, el valor del registro ESP es decrementado en 4, lo que produce que apunte hacia una dirección inferior de memoria. Es por esto que el stack de un programa se encuentra generalmente en direcciones altas de memoria.

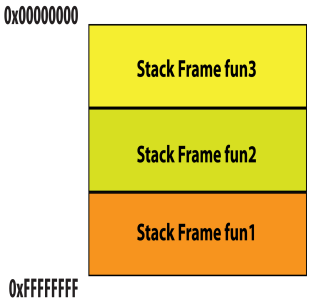
2.2.2. Funciones

2.2.2.1. Almacenamiento de variables locales

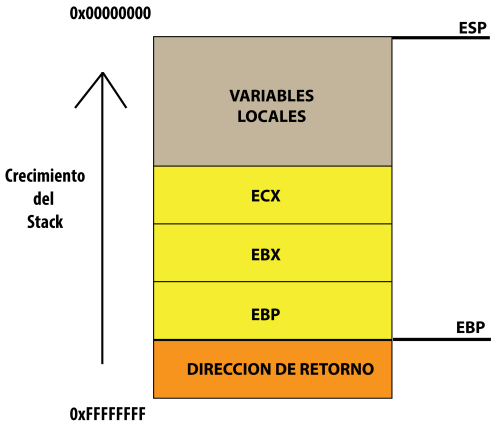
Antes de explicar como se realiza el almacenamiento de las variables locales de una función es necesario explicar el concepto de stack frame.

Un stack frame es la zona de memoria del stack que utiliza una función mientras no ha finalizado su ejecución. Es posible que durante la ejecución de una función se realicen llamadas a otras funciones, por lo que generalmente tendremos varios stack frames al mismo tiempo en el stack.

Por ejemplo, si tenemos funciones fun1, fun2 y fun3, tal que fun1 llama a fun2 y fun2 llama a fun3, el estado del stack durante la ejecución de fun3 se vería como en la Figura 2.3.a .



(a) Llamadas anidados de funciones en el stack



(b) Stack frame visto por dentro luego de la ejecución del prólogo.

Figura 2.3: Stack frame

Ahora explicaremos como es un stack frame por dentro. Cuando comienza la ejecución de una función, las primeras instrucciones de la función se encargan de crear el stack frame, este conjunto de instrucciones se denomina prólogo de la función.

El prólogo de una función se encarga de:

- Preparar el registro EBP, que será el que apunte a la base del stack frame actual.
- Salvar en el stack los valores de los registros que la función va a utilizar.
- Hacer espacio para las variables locales

```
1 PUSH EBP
2 MOV EBP, ESP
3 PUSH EBX
4 PUSH ECX
5 SUB ESP, 100h
```

Figura 2.4: Prólogo de una función.

En la Figura 2.4 podemos apreciar como se vería el prólogo de una función en ensamblador. Las instrucciones de la líneas 1, 3 y 4 son las encargadas de

guardar los valores de los registros que serán modificados durante la ejecución de la función. La instrucción de la línea 2 es la encargada de colocar en el registro EBP la base del stack frame actual y, finalmente, la instrucción en la línea 5 es la encargada de reservar el espacio para las variables locales de la función, en este caso se están reservando 256 bytes.

Generalmente los compiladores de lenguaje ensamblador soportan la escritura de números en distintas bases. La sintaxis puede variar de compilador a compilador, pero es común utilizar prefijos o sufijos que indiquen la base, por ejemplo usar la letra H, tanto mayúscula como minúscula para hexadecimal, la letra O para octal, etc. En el caso de la instrucción de la línea 5 de la Figura 2.4 podemos ver el número 256 decimal escrito en hexadecimal utilizando como posfijo una h.

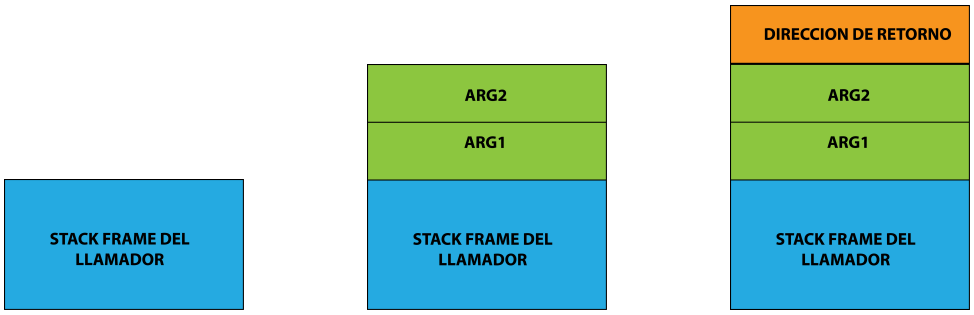
En la Figura 2.3.b se observa el estado del stack luego de la ejecución del prólogo.

Además de las variables locales y registros guardados, el stack frame de una función esta compuesto por los argumentos pasados a la función y por la dirección de retorno, ya que como dijimos anteriormente, la otra función del stack es la de intervenir en el llamado de funciones.

2.2.2.2. Llamado de funciones

Cuando se va a realizar una llamada a una función, los argumentos de la misma son colocados en el stack, así como también la dirección de retorno de la función.

Veamos como es el proceso de llamado de una función.



(a) Estado del stack antes de la llamada.

(b) Estado del stack luego del push de los argumentos

(c) Estado del stack luego del CALL

Figura 2.5: Estado del stack durante una llamada a función.

En la Figura 2.5.a podemos ver el estado del stack justo antes de realizarse

el llamado a una nueva función. Podemos ver que solamente tenemos el stack frame de la función llamadora y posiblemente otros stack frames de funciones que hayan llamado al llamador.

Al iniciarse un llamado a función, lo primero que sucede es la colocación de los argumentos de la función en el stack como se muestra en la Figura 2.5.b. Una vez que los argumentos fueron pusheados en el stack, la instrucción CALL coloca la dirección de retorno en el stack y transfiere la ejecución del programa a la nueva función. Podemos ver el resultado en la Figura 2.5.c.

Al comenzar la ejecución de la nueva función, se ejecuta el prólogo de la función. Luego se ejecutan las instrucciones propias de la función en cuestión y finalmente se ejecuta el epílogo de la función. El epílogo de la función es un conjunto de instrucciones que tiene como objeto revertir los cambios que se realizaron en el stack por el prólogo. El epílogo consiste en:

- Restaurar el valor de ESP antes del llamado a la función (almacenado en EBP)
- Restaurar el valor de los registros antes del llamado a la función.
- Devolver el control de la ejecución a la función llamadora

```
1. MOV ESP, EBP
2. POP EBP
3. RET
```

Figura 2.6: Código ensamblador del epílogo.

En la Figura 2.6 podemos ver el epílogo de una función en ensamblador. La primera instrucción restaura el valor de ESP al de antes de comenzar la ejecución de la función. Este valor había sido almacenado en EBP y no es modificado durante la ejecución de la función. La segunda instrucción restaura el valor de EBP antes de ejecutarse la función, ya que recordemos que en el prólogo de la función este valor había sido pusheado en el stack. Si la función hubiese utilizado otros registros, tendríamos una instrucción POP por cada uno de los registros.

Finalmente, la tercera instrucción toma el valor de memoria a la que apunta ESP y realiza un salto incondicional. Recordemos que cuando fue llamada la función en cuestión, la instrucción CALL coloco en el stack la dirección de retorno del llamador.

Como podemos ver, el orden en el que los datos son colocados y recuperados del stack es fundamental para lograr un correcto llamado a funciones.

Cualquier leve modificación de ESP puede suponer que a la hora de ejecutarse la instrucción RET se produzca un salto hacia un lugar distinto de la dirección de retorno de la función llamadora.

Un manejo incorrecto de las variables locales del stack nos permite sobrescribir la dirección de retorno con un valor arbitrario, pudiendo así controlar el flujo de control del programa. Este tipo de errores de programación son denominados buffer overflow basados en stack. En la próxima sección nos explayaremos más sobre los mismos.

2.2.3. Buffer overflow

Cuando los programadores manejan datos controlados por el usuario sin tener en cuenta su tamaño, generalmente en operaciones de copiado de datos, existe la posibilidad de que el código escrito sea vulnerable a buffer overflow.

Un buffer overflow se produce cuando se tratan de colocar más datos de los que un buffer puede contener o cuando se escriben datos por fuera del buffer (recordemos que el stack crece hacia direcciones inferiores de memoria). Si el buffer se encuentra en el stack, nos encontramos ante un buffer overflow basado en stack.

El peligro de este tipo de errores es que su explotación puede producir comportamientos no deseados como denegación de servicio y ejecución de código arbitrario entre otros.

2.2.3.1. Buffer overflow en C

Los buffer overflow son uno de los errores de seguridad más comunes en lenguajes que permiten que el programador controle la memoria, por ejemplo C/C++.

Como dijimos anteriormente, los buffer overflow más comunes se producen al realizar copiado de datos, sin tener en cuenta el tamaño de los mismos.

```
1 int main(int argc, char *argv[]) {
2         char buffer[4];
3         int fd = 0;
4         fd = open("test.txt",O_RDONLY);
5         int count = read(fd,buffer,28);
6         return 0;
7 }
```

Figura 2.7: Código vulnerable a buffer overflow en C.

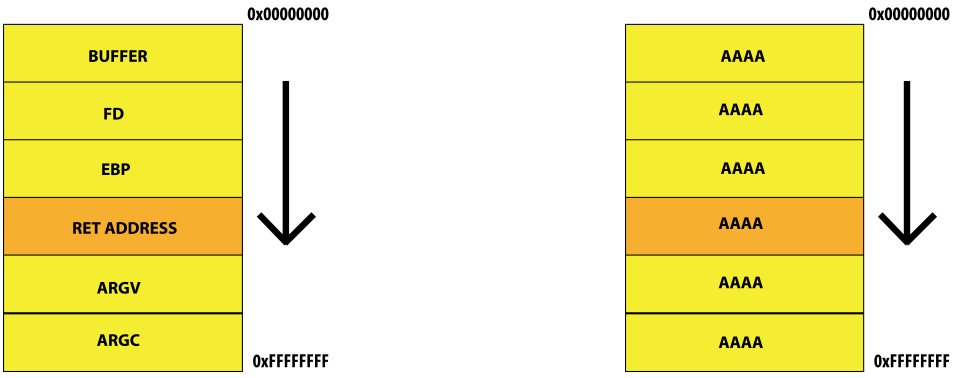
Como podemos ver en el código de la Figura 2.7, en las líneas 2 y 3 tenemos dos variables locales declaradas, de las cuales una es un buffer de 4 bytes. Recordemos que las variables locales a una función son almacenadas en el stack.

En la línea 5 podemos ver que se realiza una operación de copiado de datos hacia nuestro buffer. Notemos que la función read va a copiar 28 bytes del archivo `test.txt` en nuestro buffer de 4 bytes ya que no existe ningún chequeo previo sobre el tamaño del buffer. Esto producirá un buffer overflow en el stack.

2.2.3.2. Explotación

La explotación exitosa de un buffer overflow nos permitirá controlar la ejecución del programa vulnerable, veamos en que consiste la explotación del programa de la Figura 2.7.

En la Figura 2.8 podemos observar lo que sucede en el stack cuando se produce el buffer overflow. En la Figura 2.8.a vemos el estado del stack antes de que suceda el buffer overflow. En este punto ya se realizó el llamado a la función main, y ya se reservó el espacio para las variables locales de esta función. Como el buffer solo puede almacenar 4 bytes, y se están copiando 28 bytes, lo que sucede es que los 24 bytes restantes son copiados en la memoria adyacente al buffer, sobrescribiendo los datos que ya se encontraban allí. El copiado de los datos se realiza desde direcciones de memoria inferiores hacia direcciones de memoria superiores, a diferencia con el stack que crece hacia direcciones de memoria más bajas. En este caso estamos suponiendo que el archivo test.txt contiene solo 'A', por eso en la Figura 2.8.b vemos que los nuevos valores en el stack son 'A's. Además de sobrescribir el valor de las otras variables locales, el valor guardado de EBP y los argumentos de la función, también estamos sobrescribiendo la dirección de retorno. Esto producirá que una vez que se haya terminado la ejecución de la función y se ejecute la instrucción RET, el valor de EIP sea 0x41414141 (valor hexadecimal de 'AAAA', cada letra esta codificada en ASCII por lo que es necesario solo 1 byte para representarla).



(a) Stack antes de la operación de copiado

(b) Stack luego de la operación de copiado

Figura 2.8: Estado del stack en un buffer overflow.

Los bytes que están sobrescribiendo la dirección de retorno de la función son los bytes 13 a 16 de test.txt. Es decir, que si colocamos en test.txt:

AAAAAAAAAABBBBAAAAAA....., habremos sobrescrito la dirección de retorno con 0x42424242 (valor hexadecimal de 'BBBB'). Podemos ver así, que remplazando BBBB con el valor que deseemos podes controlar la ejecución del programa.

2.3. Satisfability Module Theories

Los problemas de decisión son problemas que pueden ser formulados como preguntas cuya respuesta puede ser afirmativa, o negativa. Un ejemplo conocido de este tipo de problemas es el de la satisfacibilidad de una fórmula proposicional (SAT). Este problema responde a la pregunta de si dada una fórmula en lógica proposicional, existe una asignación de las variables que haga que la fórmula sea verdadera.

El problema de la satisfacibilidad proposicional es un problema muy conocido y estudiado por lo que existen numerosos programas encargados de determinarla. Además de determinar la satisfacibilidad, estos algoritmos devuelven un modelo de la fórmula, es decir, una asignación de las variables de la fórmula que la hace verdadera.

SMT es una extensión del problema de SAT para formulas lógicas que permite el uso de otras teorías además de la lógica proposicional. Podemos pensarlo como una instancia de SAT en donde las variables proposicionales booleanas fueron remplazadas por términos de otras teorías. Algunos ejemplos de las teorías que se soportan en la actualidad son la lógica ecuacional, aritmética lineal, arreglos, vectores de bits, etc. o alguna combinación de las anteriores. Estas teorías nos permiten modelar problemas de un modo mucho más sencillo e intuitivo que si tuviéramos que hacer el modelado análogo utilizando SAT, extendiendo así, la expresividad de la lógica proposicional.

Con el pasar de los años se han desarrollado dos formas para determinar la satisfacibilidad de una fórmula en alguna de las teorías antedichas. El enfoque más simple, denominado *eager,* consiste en codificar la fórmula de la teoría en cuestión, en una forma equisatisfacible en lógica proposicional, de modo de poder utilizar los procedimientos de decisión SAT para determinar su satisfacibilidad. Este enfoque tiene el beneficio de que el problema de SAT ha sido muy estudiado y que con el paso de los años ha habido un gran desarrollo de los SAT solvers [22, Orbe]. El problema que posee este enfoque es que las teorías se restringen a aquellas que pueden traducirse a lógica proposicional.

El otro enfoque, denominado *lazy,* es el más utilizado en la actualidad y consiste en combinar un SAT solver junto con procedimientos de decisión específicos para cada una de las teorías en cuestión. El beneficio de este enfoque es que al usar procedimientos específicos para cada teoría es posible utilizar algoritmos y estructuras de datos específicos para los problemas del dominio en cuestión que nos permiten incrementar la eficacia de todo el sistema [22].

Las teorías que con mayor naturalidad permiten modelar el comportamiento de programas binarios son la teoría de los vectores de bits y la teoría

de los arreglos.

2.3.1. Teoría de los Vectores de Bits de tamaño fijo.

Un vector de bits es un vector, en el que cada uno de sus elementos es un bit. Trabajaremos con vectores de bits con un tamaño fijo ya que las teorías con una cantidad variable de bits son indecidibles [19].

Esta teoría surge inicialmente para realizar verificaciones formales sobre implementaciones de hardware, pero su uso es adecuado también para modelar la ejecución de código binario, ya que los registros y los datos almacenados en la memoria de un programa pueden ser considerados como vectores de bits.

Las operaciones sobre vectores de bit que la mayoría de los solvers soporta pueden dividirse en cuatro categorías:

2.3.1.1. Operadores a nivel de palabra.

En esta categoría tenemos operadores de concatenación, para unir dos bit vectors en uno más grande, extracción, para obtener un determinado rango de bits dentro de un bit vector, shift para la izquierda y derecha, rotación para la izquierda y derecha, extensión con signo.

2.3.1.2. Operadores aritméticos.

Notemos que los operadores aritméticos para vectores de tamaño fijo n, tienen la peculiaridad que toda la aritmética se realiza módulo n.

A la hora de realizar operaciones aritméticas con vectores de bits tenemos que tener en cuenta que existen tanto operaciones signadas como no signadas. Esto implica que al realizar una operación aritmética con un operador signado, los vectores de bits utilizados en la operación serán pensados como si estuvieran representando un número en complemento dos. Es por esto que es posible obtener distintos resultados al utilizar los mismos vectores pero utilizando un operador signado en un caso, y uno sin signo en el otro.

Los operadores aritméticos son: suma, resta, multiplicación, división, módulo y opuesto, tenemos versiones signadas y no signadas para cada uno de ellos.

2.3.1.3. Operadores bit a bit.

Los operadores bit a bit son: and, or, xor, not, nand, nor.

2.3.1.4. Operadores de comparación.

Los operadores de comparación son =, <, <=, >, >=.

El único operador que no tiene dos versiones (signada y sin signo) es el operador de igualdad =. Dos vectores son iguales si cada uno de sus bits son iguales.

Para el resto de los operadores tenemos versiones con y sin signo. Los operadores de comparación son los mismos de la aritmética tradicional pero usando números codificados en binario (sin signo) y complemento dos (con signo).

2.3.2. Teoría de los arreglos.

Un arreglo es un mapeo entre un índice y un valor donde tanto índice como valor pueden ser de distintos tipos. Los arreglos soportan dos operaciones fundamentales: lectura y escritura.

El operador de lectura toma dos parámetros, un arreglo y una posición. Así, *lectura(a,i)* significa leer lo que hay en la posición i del arreglo a.

El operador de escritura toma tres parámetros, un arreglo, una posición y un dato. Así, *escritura(a,i,v)* significa escribir en la posición i del arreglo a el valor v.

Los siguientes axiomas caracterizan la teoría de los arreglos [19].

$\forall a.\forall i.\forall e.(lectura(escritura(a,i,e),i) = e)$

$\forall a.\forall i.\forall j.\forall e.((i \neq j) \rightarrow lectura(escritura(a,i,e),j) = lectura(a,j))$

$\forall a.\forall b.(\forall i.(lectura(a,i) = lectura(b,i)) \rightarrow (a = b))$

Los dos primeros axiomas caracterizan a las operaciones de lectura y escritura de un arreglo. El primer axioma nos dice que si escribimos en una posición del arreglo un valor e, y luego leemos esa posición del arreglo, obtendremos nuevamente e. El segundo dice que si realizamos una escritura en la posición j de un arreglo, ninguno de las otras posiciones del arreglo se vera modificada.

Por último, el tercer axioma define cuando podemos decir que dos arreglos son iguales, esto es, cuando para cada posición dentro del arreglo ambos arreglos almacenan el mismo valor.

Esta teoría nos permite modelar adecuadamente la idea de memoria de un programa ya que podemos pensar la memoria de una computadora como un gran arreglo de bytes. Las únicas operaciones que realizamos sobre la memoria son leer y escribir datos, por lo que las operaciones fundamentales de un arreglo se adecuan naturalmente al funcionamiento básico de la memoria.

El tipo del índice y valores que almacenaremos en la memoria dependerán de la arquitectura de hardware que queramos modelar. En la arquitectura Intel x86, los índices serán serán vectores de 32 bits, mientras que los valores vectores de 8 bits.

2.3.3. Otras teorías.

Otras teorías que son muy populares pero que no usaremos en esta tesis son: la teoría de la lógica ecuacional, la teoría de la aritmética lineal y aritmética no lineal.

La teoría de la lógica ecuacional es como como la teoría de la lógica proposicional pero remplazando las variables booleana por igualdades entre variables que pueden ser números reales o enteros.

La teoría de la aritmética linear y aritmética no lineal, nos permiten modelar ecuaciones lineales y no lineales respectivamente.

Es posible extender cualquiera de estas lógicas utilizando cuantificadores universales y existenciales.

2.4. Binary Analysis Platform (BAP)

BAP es un framework que nos facilita el análisis de código binario. A grandes rasgos esta dividido en dos partes, front-end y back-end. El front-end lee las instrucciones de ensamblador de un binario y luego convierte esas instrucciones a un lenguaje intermedio. El back-end es donde se implementan los algoritmos de análisis y verificación de programas, estos algoritmos trabajan sobre el código intermedio. La última versión de BAP también nos permite trabajar sobre trazas de ejecución de un programa. Estas trazas pueden ser generadas por una pintool que viene por defecto con BAP. Una pintool es un programa que utiliza el framework de instrumentación binaria PIN de Intel [9], el cual nos permite agregar código dinámicamente para que se ejecute junto con el código original.

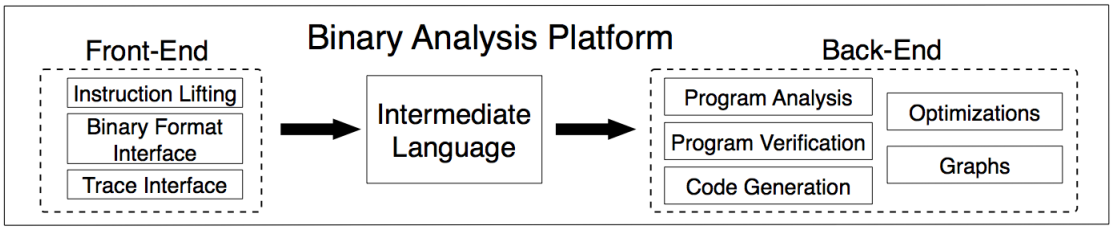


Figura 2.9: Arquitectura de BAP. [15]

2.4.1. El lenguaje intermedio

Para ver la importancia de utilizar un lenguaje intermedio a la hora de hacer análisis sobre código binario, tomemos como ejemplo la instrucción de ensamblador ADD EAX, 0x5, la cual incrementa en 5 el valor almacenado en el registro EAX.

Algunas instrucciones ensamblador, además de modificar los registros involucrados directamente en la instrucción, modifican el registro de flags. En nuestro ejemplo, el único registro modificado directamente es EAX y los flags modificados son los de overflow, sign, zero, auxiliary carry flag, parity y carry. Realizar un análisis manual de cada una de las instrucciones de un programa se convierte entonces en una tarea lenta y propensa a errores. Un correcto análisis de la modificación del registro de flags es importante ya que la semántica de muchas instrucciones, por ejemplo las instrucciones de control, dependen del estado de los flags.

Si a esto le sumamos que el set de instrucciones ensamblador de los procesadores más modernos cuenta con muchísimas instrucciones, y, que a medida

que van evolucionando los procesadores se van agregando nuevas instrucciones, es de esperar que los investigadores tratando de realizar análisis sobre código binario pasen más tiempo analizando la modificación de los registros y flags de cada instrucción que desarrollando algoritmos de análisis en sí.

Por estas razones, utilizar un lenguaje intermedio nos permite evitar la tediosa, y propensa a errores tarea de tener que analizar como modifican los registros de uso general y registro de flags las numerosas instrucciones del set de instrucciones del lenguaje ensamblador.

El uso de un lenguaje intermedio para representar la semántica del lenguaje ensamblador ya ha sido utilizado con anterioridad por varios investigadores. [18, 14, 16]. Lo que todos estos lenguajes tienen en común es que están compuestos de unas pocas instrucciones que permiten expresar correctamente la semántica de las instrucciones de ensamblador.

BIL (BAP Intermediate Language) esta compuesto por solo 7 instrucciones:

- Asignación
- Salto condicional
- Salto incondicional
- Label
- Halt
- Assert
- Special

Más adelante explicaremos cada una de las instrucciones. A su vez cada instrucción puede estar compuesta por una o más expresiones de diferentes tipos. En la Figura 2.10 podemos observar la sintaxis del lenguaje intermedio de BAP.

Las expresiones más importantes que soporta BAP son:

- Lectura de datos de arreglos
- Almacenamiento de datos en arreglos
- Expresiones binarias
- Expresiones unarias
- Constantes

program	::=	*stmt**
stmt	::=	*var* := *exp* | `jmp`(*exp*) | `cjmp`(*exp*,*exp*,*exp*)
		| `halt`(*exp*) | `assert`(*exp*) | `label` *label_kind* | `special`(string)
exp	::=	`load`(*exp*, *exp*, *exp*, τ_{reg}) | `store`(*exp*, *exp*, *exp*,*exp*,τ_{reg}) | *exp* $\Diamond_b$ *exp*
		| $\Diamond_u$ *exp* | *var* | `lab`(string) | *integer* | `cast`(*cast_kind*,τ_{reg},*exp*)
		| `let` *var* = *exp* `in` *exp* | `unknown`(string, τ) | `name`(*exp*)
label_kind	::=	*integer* | string
cast_kind	::=	`unsigned` | `signed` | `high` | `low`
var	::=	(string, id_v, τ)
$\Diamond_b$	::=	$+, -, *, /, /_s, \text{mod}, \text{mod}_s, \ll, \gg, \gg_a, \&, \mid, \oplus, ==, !=, <, \leq, <_s, \leq_s$
$\Diamond_u$	::=	$-$ (unary minus), $\sim$ (bit-wise not)
value	::=	*integer* | *memory* | string | $\perp$
integer	::=	n ($:\tau_{\text{reg}}$)
memory	::=	{ *integer* $\rightarrow$ *integer*, *integer* $\rightarrow$ *integer*, ... } ($:\tau_{\text{mem}}$)
τ	::=	$\tau_{reg} \mid \tau_{mem}$
τ_{mem}	::=	`mem_t`(τ_{reg}) | `array_t`($\tau_{\text{reg}}, \tau_{\text{reg}}$)
τ_{reg}	::=	`reg1_t` | `reg8_t` | `reg16_t` | `reg32_t` | `reg64_t`

Figura 2.10: Sintaxis de BIL.[15]

- Variables
- Casteos de tipo

Notar que a simple vista la expresión "Almacenamiento de datos" puede considerarse como una instrucción. Como se explicará más adelante, este tipo de construcción no tiene efectos secundarios sobre la memoria representada, y es por ello que en la terminología de BAP se las denomina expresiones.

En la Figura 2.11 podemos ver el lenguaje intermedio de la instrucción de ensamblador ADD EAX,2. En la mayoría de los casos, una instrucción ensamblador equivale a varias instrucciones del lenguaje intermedio. Para diferenciar entre distintas instrucciones ensamblador se utiliza, en el lenguaje intermedio, un label que referencia a la dirección en la que se encontraba esa instrucción en el binario original. Esto podemos apreciarlo en la Figura 2.11. La primera instrucción es un label que referencia a la dirección 0, indicando que estamos viendo el código intermedio de la primera instrucción del binario analizado.

Tanto la primera, como la segunda instrucción en la Figura 2.11, líneas 1 y 2, son labels (sección 2.4.1.3), el resto de las instrucciones son asignaciones (sección 2.4.1.3).

Todas las instrucciones pueden tener, opcionalmente, atributos que nos proveen de información adicional sobre la instrucción en cuestión. Los atributos se encuentran a continuación de la instrucción del lenguaje intermedio en

```
1  addr 0x0 @asm "add    $0x2,%eax"
2  label pc_0x0
3  T_t1:u32 = R_EAX:u32
4  T_t2:u32 = 2:u32
5  R_EAX:u32 = R_EAX:u32 + T_t2:u32
6  R_CF:bool = R_EAX:u32 < T_t1:u32
7  R_OF:bool = high:bool((T_t1:u32 ^ ~T_t2:u32) & (T_t1:u32 ^ R_EAX:u32))
8  R_AF:bool = 0x10:u32 == (0x10:u32 & (R_EAX:u32 ^ T_t1:u32 ^ T_t2:u32))
9  R_PF:bool =    ~low:bool(R_EAX:u32 >> 7:u32 ^ R_EAX:u32 >> 6:u32 ^
10          R_EAX:u32 >> 5:u32 ^ R_EAX:u32 >> 4:u32 ^
11          R_EAX:u32 >> 3:u32 ^ R_EAX:u32 >> 2:u32 ^
12          R_EAX:u32 >> 1:u32 ^ R_EAX:u32)
13 R_SF:bool = high:bool(R_EAX:u32)
14 R_ZF:bool = 0:u32 == R_EAX:u32
```

Figura 2.11: Lenguaje intermedio para la instrucción ADD. Podemos apreciar la modificación de los flags.

cuestión. Cada atributo comienza con un @ seguido del nombre del atributo. Es posible que haya más de un atributo por instrucción. La sintaxis para los atributos de las instrucciones es la siguiente:

```
instrucción @attr1 attr1_info ..... @attrn attrn_info
```

Por ejemplo, en la primera línea de la Figura 2.11, podemos ver, como dijimos anteriormente, una instrucción de tipo label que referencia una dirección. Esta instrucción tiene un atributo de tipo asm con un string que representa la instrucción ensamblador.

Algunos de los atributos del lenguaje intermedio son:

- @asm: contiene un string con una instrucción en ensamblador.
- @str: contiene un string cualquiera.

2.4.1.1. Tipos básicos

BAP soporta dos tipos básicos de datos: valores numéricos y arreglos.

El tipo de una expresión se determina por lo que sigue de los dos puntos (:).

```
exp:tipo
```

La sintaxis para declarar el tipo de los valores numéricos es:

Sintaxis	Tipos numéricos
bool	valor numérico de 1 bit
u8	valor numérico de 8 bits
u16	valor numérico de 16 bits
u32	valor numérico de 32 bits
u64	valor numérico de 64 bits

Las arreglos son mapeos entre números similares a los que podemos encontrar en el lenguaje C. La sintaxis para el tipo arreglo es:

Sintaxis	Tipos de arreglos
?bool	arreglo de 1 bit
?u8	arreglo de 8 bit
?u16	arreglo de 16 bit
?u32	arreglo de 32 bit
?u64	arreglo de 64 bit

Lo que diferencia un tipo numérico de un arreglo es básicamente la presencia del símbolo ?, antes de indicar la cantidad de bits. Que un arreglo sea de n bits quiere decir que el índice que usaremos para acceder a los datos del arreglo es un número de n bits.

Los valores de las expresiones de BAP pueden ser numéricos o arreglos. La única expresión que tiene como resultado un valor de tipo arreglo es el "Almacenamiento de datos", todas las demás expresiones dan como resultado un número. Los arreglos serán utilizados para modelar la memoria de los programas.

A continuación veremos en detalle las expresiones e instrucciones del lenguaje intermedio. Comenzaremos con las expresiones ya que las instrucciones están compuestas por estas.

2.4.1.2. Expresiones

Constantes Las constantes son expresiones de tipo numérico, que se utilizan para representar un número de una cantidad fija de bits. La cantidad de bits esta determinada por el tipo de la expresión.

La sintaxis es:

```
value : int_type
```

La cantidad de bits de la constante está determinada por *int_type*, que es uno de los tipos básicos numéricos definidos en 2.4.1.1. *value* es el número a representar, puede ser escrito en decimal o en hexadecimal. Para que el número sea considerado en base hexadecimal es necesario anteponer 0x al mismo.

Por ejemplo, para definir el entero 0x100 con 32 bits, la sintaxis sería:

0x100 : u32

En caso de que value sea un número de más de int_type bits, solamente se tomaran los int_type bits menos significativos de value. Por ejemplo,

0x1234 : u8

es equivalente a

0x34 : u8

En la Figura 2.11 podemos ver enteros en las líneas 4,8,9,10,11,12 y 14.

Expresiones Unarias Las expresiones unarias están conformadas por un operador, que determina el tipo de expresión unaria y por una expresión de tipo numérico (en caso contrario se producirán errores de tipo). Estas expresiones pueden ser aritméticas (número negativo) o lógicas (negación bit a bit), los operadores asociados son - y ~ respectivamente y devuelven un valor de tipo numérico. El tipo de la expresión devuelta será el tipo de la expresión operando.

La sintaxis es:

operador expr

Por ejemplo, si tenemos la expresión de tipo u32:

0x123 : u32

el resultado de la expresión unaria de negación aritmética:

−(0x123 : u32)

también tendrá como tipo u32.

En la Figura 2.11 podemos ver expresiones unarias de negación bit a bit en las líneas 7 y 9.

Expresiones Binarias Las expresiones binarias están formadas por dos expresiones de tipo numérico y por un operador que determina el tipo de expresión binaria. Estas expresiones pueden ser aritméticas o lógicas, los posibles operadores son:

Operador	Tipo
+	Adición
-	Sustracción
*	Multiplicación
/	División
$/	División con signo
%	módulo
$ %	módulo con signo
<<	Shift hacia la derecha
>>	Shift hacia la izquierda
$>>	Shift hacia la izquierda con signo
&	And Lógico
|	Or Lógico
^	Xor Lógico
==	Igualdad
¡¿	Desigualdad
<	Menor
<=	Menor igual
$<	Menor con signo
$<=	Menor igual con signo

La sintaxis es:

```
expr1 operador expr2
```

Es necesario que ambas expresiones (expr1 y expr2) sean de la misma cantidad de bits, caso contrario obtendremos errores de tipo. El resultado de esta expresión es un valor numérico de la misma cantidad de bits que las expresiones operandos.

Por ejemplo para sumar enteros 1 y 2 la sintaxis sería:

```
1:u32 + 2:u32
```

En la Figura 2.11 podemos ver operadores binarios en las líneas 5(+), 6(<), 7,8(&,^),9,10,11,12(>>, ^), 14(==).

Variables Las variables nos permiten almacenar datos de distintos tipos.

La sintaxis es:

```
var_name:var_type
```

donde var_type puede ser de tipo numérico o arreglo.

Estas variables de tipo numéricas nos permiten modelar los registros de una computadora.

Por ejemplo, para declarar la variable foo de tipo numérico de 32 bits la sintaxis sería:

FOO: u32

Para declarar un arreglo con nombre mem, que almacenará valores de 32 bits, la sintaxis sería:

mem: ? u32

En la Figura 2.11 podemos ver variables en todas las líneas salvo las dos primeras, algunas de las variables son R_EAX, T_t1, T_t2, R_OF, etc y en la líneas 3 y 6 de la Figura 2.12 en la página 34, podemos ver una variable de tipo arreglo.

Casteos Los casteos se utilizan para manejar los distintos tipos de direccionamiento de memoria que soporta el lenguaje ensamblador y para realizar operaciones aritméticas o lógicas con valores de tipo numérico de distinta cantidad de bits. Las expresiones de casteo son las únicas expresiones del lenguaje intermedio cuyo resultado puede tener un tipo con una cantidad de bits distinta que los operandos.

Su sintaxis es:

```
cast_type:cast_size(expr)
```

- cast_type: puede ser high, low, signed, unsigned.
- cast_size: uno de los tipos numéricos bool, u8, u16 o u32.
- expr: expresión numérica.

Algunos ejemplos de casteo son:

1. `high:u8(0xcafecafe:u32)`
2. `low:bool(R_EAX:u32)`
3. `signed:u32(0x10:u8)`
4. `unsigned:u32(0x10:u8)`

El primer casteo (high) es igual a tomar los 8 bits más significativos del número 0xcafecafe almacenado como un entero de 32 bits. El segundo casteo (low) es igual a tomar el bit menos significativo de la variable R_EAX de 32 bits. El tercer casteo se usan para incrementar la cantidad de bits de un número binario sin perder el signo del número, mientras que el cuarto casteo hace lo mismo pero sin preservar el signo del número. Para más información sobre casteos ver [4].

En la Figura 2.11 podemos ver casteos de tipo high en las líneas 7 y 13, y de tipo low en la línea 9.

```
1  addr 0x401046 @asm "mov     0x40c1e0,%eax"
2  label pc_0x401046
3  R_EAX:u32 = mem:?u32[0x40c1e0:u32, e_little]:u32
4  addr 0x401050 @asm "movl    $0x0,-0x28(%ebp)"
5  label pc_0x401050
6  mem:?u32 = mem:?u32 with [R_EBP:u32 + 0xffffffd8:u32, e_little]:u32 = 0:u32
```

Figura 2.12: Instrucciones del lenguaje intermedio con load y store.

Operaciones sobre arreglos. Como dijimos anteriormente, cuando explicábamos los posibles tipos de una variable, un programa escrito en el lenguaje intermedio de BAP puede manejar varios arreglos.

Las expresiones para trabajar sobre arreglos del lenguaje intermedio son dos: load y store. Load se utiliza para leer datos de una variable de tipo arreglo y store se utiliza para escribir datos en una variable del mismo tipo. Ninguna de las dos expresiones tiene efectos secundarios sobre la memoria representada.

Es claro que Load no tiene efectos secundarios pues solamente devuelve los datos que se encuentran almacenados en la variable de tipo arreglo, lo que no queda del todo claro es que la expresión Store tampoco tiene efectos secundarios, pero esto se debe a que esta expresión nos devuelve un valor de tipo arreglo igual al anterior pero con los datos modificados en la posición establecida. Al crearse un nuevo valor de tipo arreglo no se está modificando el estado del programa de ninguna manera, esto solo sucederá si asignamos el nuevo valor de tipo arreglo a la variable de tipo arreglo anterior.

Para clarificar este último punto, supongamos que estamos modelando un código binario donde la memoria del programa será modelada con una variable de tipo arreglo con nombre mem. Al realizar una operación de store sobre esa variable, obtendremos un nuevo valor de tipo arreglo con los datos modificados. El estado del programa no se verá modificado hasta que remplazamos la variable mem con el nuevo valor de tipo arreglo que obtuvimos de la operación store.

Store Esta expresión se utiliza a la hora de almacenar datos en un determinado índice de una variable de tipo arreglo. La sintaxis es:

```
mem_name:?mem_type with [where_exp, endianess]:op_size = what_expr
```

- mem_name: el nombre del arreglo donde se guardarán los datos.
- mem_type: indica el tamaño de los índices del arreglo a utilizar, en arquitecturas de 32 bits el valor es u32, mientras que en arquitecturas de 64 bits es u64.
- where_exp: una expresión numérica que indica donde se van a guardar los datos, debe tener el mismo tipo que mem_type.
- what_expr: una expresión numérica que indique que datos se van a guardar, debe tener el mismo tipo que op_size.
- endianess: el tipo de endianess que será utilizado para guardar los datos, puede ser e_little o e_big.
- op_size: la cantidad de bytes a copiar, debe tener el mismo tipo que what_expr.

Como ya mencionamos, es importante notar que esta expresión no modifica directamente la variable arreglo mem_name, sino que devuelve un nuevo arreglo que es idéntico al arreglo original, con excepción de la posición where_exp, el valor del arreglo en esa posición es what_expr.

Por ejemplo, para obtener una copia del arreglo mem con el valor 0xcafecafe almacenado en el índice 0x1000 (en una arquitectura de 32 bits little endian), la sintaxis sería:

```
mem:?:u32 with [0x1000:u32, e_little]:u32 = 0xcafecafe:u32
```

En la Figura 2.12, línea 6 podemos ver una asignación en donde la variable a asignar es de tipo arreglo y la expresión a asignar es un store (que devuelve un valor de tipo arreglo). Notemos que ambos hacen referencia a la misma variable arreglo, por lo que luego de ejecutarse la asignación, la variable arreglo mem contendrá un nuevo valor de tipo arreglo que será el utilizado para los usos posteriores a la asignación. Recordemos una vez más que la expresión store por si sola no modifica al arreglo mem, el arreglo se ve modificado solo cuando se produce una asignación.

Load Esta expresión se usa para obtener los datos almacenados en un determinado índice de un arreglo.

La sintaxis es:

```
mem_name:?mem_type[where_exp, endianess]:op_size
```

- mem_name: el nombre de la variable de tipo arreglo de la cual se leerán los datos.
- mem_type: indica el tamaño del índice a utilizar, en arquitecturas de 32 bits el valor es u32, mientras que en arquitecturas de 64 bits es u64.
- where_expr: una expresión numérica que indica el índice desde el cual se leerán los datos, debe tener el mismo tipo que mem_type.
- endianess: el tipo de endianess que será utilizado para leer los datos, puede ser e_little o e_big.
- op_size: la cantidad de bytes a copiar.

Por ejemplo, para leer solo 8 bits del contenido del arreglo en el índice 0x12345 (en una arquitectura de 32 bits little endian), del arreglo con nombre mem la sintaxis sería:

```
mem:?u32[0x12345:u32, e_little]:u8
```

En la Figura 2.12, línea 3 podemos ver una expresión de tipo load que carga los datos leídos en la variable R_EAX.

2.4.1.3. Sintaxis de las Instrucciones

Asignación Una asignación esta compuesta por una variable y una expresión, la evaluación de la expresión es almacenada en la variable. Para evitar errores de tipo es necesario que tanto la variable como la expresión a asignar sean del mismo tipo. Las líneas 3 a 14 de la Figura 2.11 son asignaciones. La sintaxis es:

```
var:type = expr
```

Salto Incondicional Toma un label y transfiere la ejecución del programa a ese label. La sintaxis es:

```
jmp "label"
```

Salto Condicional Toma una expresión y dos labels, uno para cuando la expresión evaluada es verdadera y otro para cuando la expresión es falsa. Transfiere la ejecución del programa al label que corresponda. La sintaxis es:

```
cjmp bool_expr, "label_true", "label_false"
```

Label Los labels son identificadores que pueden hacer referencia a una dirección, como el label de la línea 1 en la Figura 2.11, o a un nombre, como el label de la línea 2 de la misma figura.

En el primer caso la sintaxis es:

```
addr address_number
```

Mientras que en el segundo caso, la sintaxis es:

```
label label_name
```

Como dijimos anteriormente, las dos primeras líneas la Figura 2.11 son labels.

Halt Termina la ejecución del programa. La sintaxis es:

```
halt
```

Assert Esta instrucción es similar al assert de C, toma una expresión de tipo booleano. Si la expresión evaluada es falsa, se da por terminada la ejecución del programa. La sintaxis es:

```
assert bool_expr
```

Special Esta instrucción se utiliza para realizar llamadas a funciones externas, como por ejemplo system calls. La semántica de este tipo de instrucciones no esta definida, sino que se define a la hora de realizar el análisis del programa.

Comentario Los comentarios de BAP siguen el mismo formato que los comentarios de C. El código a comentar se encierra entre */ y /*.

```
/* Comentario en el medio */
```

2.4.1.4. Semántica Operacional

El lenguaje intermedio de BAP tiene una semántica operacional bien definida, para más información sobre la semántica revisar [15].

2.4.2. Transformaciones necesarias

Las transformaciones que nos serán de utilidad para esta tesis y que serán explicadas a continuación son:

- Conversión a SSA (la cual explicaremos más adelante) y eliminación de código muerto y redundante. Si bien no utilizaremos estas transformaciones de forma directa, BAP las utiliza por defecto para optimizar los análisis.
- Conversión a grafo (solo para chequear visualmente que nuestras modificaciones al lenguaje intermedio son correctas).
- Loop Unrolling.
- Calculo de la precondición más débil.

2.4.3. Funcionamiento

BAP esta compuesto por un conjunto de programas que nos permitirán trabajar sobre el código binario. A continuación explicaremos brevemente cual es la función de cada uno de ellos. En la Figura 2.13 podemos ver como se relacionan estos programas entre si. El front-end de BAP está compuesto solo por la utilidad `toil`, mientras que el back-end esta compuesto por los programas `iltrans`, `toformula`, `evaluator` y `toc`.

2.4.3.1. toil

Esta utilidad es la encargada de tomar un archivo de código binario y convertirlo al lenguaje intermedio de BAP. Primero convierte el código binario a instrucciones ensamblador y luego convierte las instrucciones al código intermedio.

La utilidad soporta dos algoritmos de desensamblado. El primero es un algoritmo de desensamblado lineal, esto quiere decir que los bytes del código son convertidos a instrucciones ensamblador secuencialmente. Para esto es necesario especificarle el rango dentro del binario que deseamos desensamblar mediante dos offsets, uno de inicio y otro de fin.

El otro algoritmo de desensamblado soportado es uno recursivo descendente. Este algoritmo toma como entrada el binario y el offset a partir del cual queremos correr el algoritmo. El algoritmo realiza un desensamblado del código similar al algoritmo lineal, con la excepción de que cada vez que encuentra un salto directo, por ejemplo un CALL, el algoritmo también será aplicado recursivamente sobre la dirección destino del salto directo.

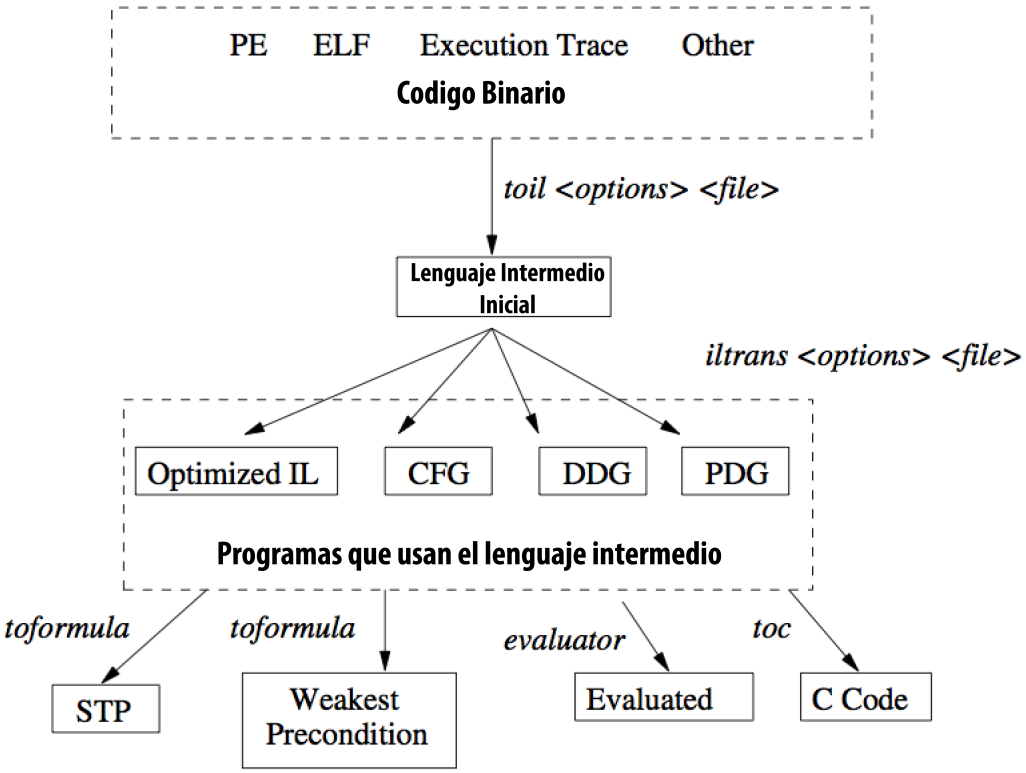


Figura 2.13: Relación entre las distintas utilidades de BAP.

También es posible realizar una transformación extra y convertir el lenguaje intermedio a otros formatos como XML [3], JSON (Sección 2.5), PROTOBUF [7].

En este trabajo utilizaremos la conversión a JSON y el algoritmo de desensamblado recursivo descendente.

Por ejemplo, para obtener el código intermedio del archivo binario test.exe, a partir del offset 0x100 utilizando un algoritmo recursivo descendente y exportar el resultado a JSON en el archivo test.json, deberíamos utilizar el programa de esta forma:

```
toil -bin test.exe -binrecurseat 0x100 -tojson -o test.json
```

2.4.3.2. iltrans

iltrans nos permite realizar transformaciones y análisis sobre el lenguaje intermedio. El funcionamiento de esta utilidad es similar al estilo de pipes & filters de Unix, se van aplicando transformaciones y el resultado de una transformación es la entrada de la siguiente, lo que nos permite aplicar varias en cadena.

Las transformaciones que nos permite realizar son:

Grafos Permite obtener grafos a partir del lenguaje intermedio. Podemos exportar estos grafos a graphviz DOT [11] para su visualización. Los grafos que soporta son:

- Grafos de control de flujo
- Grafos de dependencia de datos
- Grafos de dependencia del programa

Los grafos de control de flujo son representaciones mediante el uso de grafos dirigidos de las posibles ejecuciones que puede seguir un programa. Los nodos del grafo son conjuntos de instrucciones ensamblador que serán ejecutadas secuencialmente. Cada uno de estos nodos se denominan bloques básicos y están compuestos solo por instrucciones que no modifican el flujo de control del programa, es decir, no hay instrucciones de control dentro del conjunto. Las instrucciones de control, como los saltos condicionales, son las que determinan las aristas que conectan a los distintos nodos.

Por ejemplo, en la Figura 2.15 podemos ver el grafo de control de flujo del programa de la Figura 2.14. En este grafo podemos ver tres nodos, uno con 4 instrucciones y dos con 2 instrucciones. Podemos apreciar que la última instrucción del nodo número 1 es una instrucción de control, lo que produce

que este nodo este conectado con los nodos 2 y 3, dependiendo del resultado de la comparación de EAX con 0x2. Los grafos de dependencia de datos y de dependencia de programa no son relevantes para este trabajo, por lo que no es un tema que será desarrollado en el mismo, para más información sobre estos grafos referirse a [12, pag. 722,854].

```
        MOV EAX,EBX
        ADD EAX,0x1
        CMP EAX,0x2
        JNZ label1
        MOV EAX,0X1
        RET
label1:
        MOV EAX,0X0
        RET
```

Figura 2.14: Código ensamblador del GCF de la Figura 2.15.

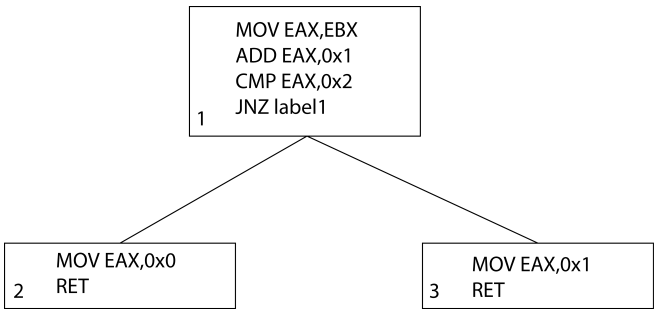


Figura 2.15: Grafo de control de flujo

Por ejemplo, para obtener el grafo de control de flujo del archivo con lenguaje intermedio main.il en formato graphviz DOT deberíamos realizar lo siguiente:

```
iltrans -il main.il -to-cfg -pp-ast-cfg main.dot
```

Muchas de las transformaciones necesitan que el lenguaje intermedio haya sido convertido a grafo de control de flujo primero para poder aplicarse. Por ejemplo, para eliminar los ciclos de un programa en el lenguaje intermedio tenemos que utilizar el siguiente comando:

```
iltrans -il main.il -to-cfg -rm-cycles -to-ast -pp-ast
    main_done.il
```

Observemos que las transformaciones se irán aplicando de izquierda a derecha. Así, primero convertimos el código en CFG utilizando -to-cfg. Después utilizamos -rm-cycles para remover los ciclos. Luego convertimos el resultado a la sintaxis abstracta de BAP usando -to-ast para finalmente para obtener el resultado en el archivo de texto main_done.il mediante -pp-ast.

SSA (Single Static Assignment) Soporta la conversión del código intermedio a SSA [2]. Al estar definida cada variable solo una vez, se simplifica el desarrollo de algoritmos de análisis.

Chopping El chopping o slicing [1] es un análisis que nos permite determinar, dado un nodo de origen y un nodo destino del grafo de control de flujo de un programa, el subgrafo del grafo original tal que las definiciones de variables en el nodo de origen, afectan a las variables usadas en el nodo de destino.

Data-Flow Este tipo de análisis consiste en determinar los posibles valores que pueden adoptar las variables de un programa en un momento dado, pero sin ejecutar el programa. Algunos de los posibles análisis son:

- Use-def: dada una variable que se use en un determinado punto del programa, este algoritmo devuelve todas las definiciones de variables en las que la variable en cuestión este presente.
- Def-use: dada una variable que este definida en un determinado punto del programa, este algoritmo devuelve todos los usos que se hagan de esta variable.
- Value Set Analysis: este algoritmo devuelve para cada punto del programa un conjunto de posibles valores que pueden adoptar las variables del programa.
- Live-variable analysis: este algoritmo permite determinar dado un punto del programa que variables serán utilizadas posteriormente por el programa.

Para más información sobre este tipo de algoritmos consultar [12].

Optimizaciones Las optimizaciones son muy importantes a la hora de generar la fórmula SMT, ya que permiten que el solver decida la satisfacibilidad de la fórmula de forma más eficiente.

- Eliminación de código muerto
- Optimizaciones sobre el flujo de control
- Eliminar nodos inalcanzables de grafos de control.
- Simpson's global value numbering: este algoritmo permite eliminar código redundante [21, sec. 12.4.2].

Loop Unrolling Los ciclos son una parte fundamental de cualquier programa pero BAP SOLO SOPORTA LA VERIFICACIÓN DE PROGRAMAS ACÍCLICOS. Es por esto que para eliminar los ciclos, una técnica utilizada con frecuencia es la de expandir los ciclos una cantidad acotada de veces. La expansión consiste básicamente en repetir una cantidad fija de veces la semántica de lo que se ejecutará dentro del ciclo. Veremos más sobre esto en el próximo capítulo.

Veamos un ejemplo del usos de iltrans para convertir el código intermedio de un programa a un grafo de control y realizar un unroll de los loops 5 veces.

```
iltrans -il main.il -to-cfg -unroll 5 -to-ast -pp-ast
    main_done.il
```

2.4.3.3. topredicate

Esta utilidad nos permite realizar tareas de verificación de los programas binarios. El argumento más importante que toma es un predicado, representado en el lenguaje intermedio, que actúa como postcondición a la hora de la verificación. El programa en lenguaje intermedio es convertido a Dijkstra's Guarded Command Language (GCL) [17] y junto con la postcondición que se le dio como argumento se calcula la precondición más débil con respecto al programa en GCL. Como la precondición más débil es una fórmula, podemos pasarle esta fórmula a un SMT solver para que nos devuelva un modelo (asignación de las variables libres de la fórmula) que satisfaga la postcondición inicial, o bien nos diga que la fórmula es insatisfacible y por lo tanto no existe tal modelo. Es posible generar formulas compatibles con CVC Lite [10], SMTLib1 y SMTLib2 [8].

El algoritmo utilizado para la conversión a GCL toma como entrada el grafo de control de flujo de un programa y produce la representación en GCL del mismo.

Como dijimos anteriormente, BAP solo permite calcular las precondiciones más débiles de programas sin ciclos, es por esto que para obtener un programa acíclico se realiza la expansión de los ciclos del mismo, siendo la

cantidad de veces que se desean expandir los ciclos uno de los argumentos del algoritmo. Para hacer esto utilizamos la utilidad iltrans como vimos en 2.4.3.2.

Una vez que tenemos el grafo de control de flujo del programa sin ciclos, podemos correr el algoritmo de la Figura 2.16 que convertirá el grafo de control a GCL.

```
grafo = expandir_ciclos(k)
for v in vértices de la transpuesta de grafo en orden topologico:
    if |succ(v)| = 0:
        g := to_gcl(v)
    elif |succ(v)| = 1:
        g := M(succ(v)); to_gcl(v)
    else:
        v1, v2 := succ(v)
        (gsuffix, g1, g2) := split_suffix(M(v1), M(v2))
        e = branch_predicate(v, v1)
        g := (assume e; g1) choice (assume not e; g2); gsuffix
    M[v] := g
```

Figura 2.16: Algoritmo para convertir el GFC de un programa en GCL.

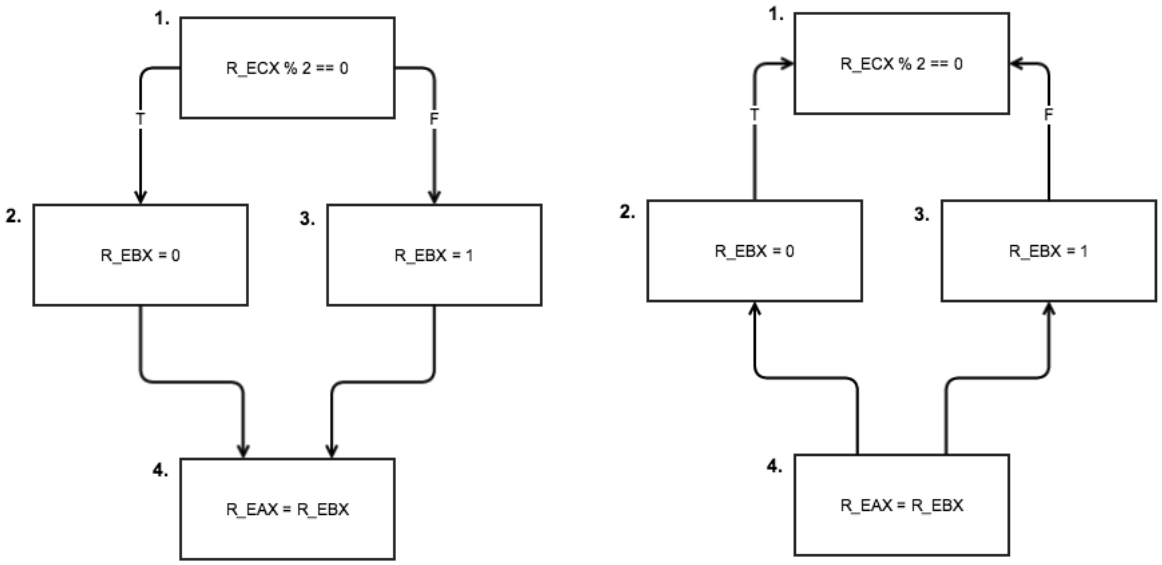
El orden en el que se recorrerán los vértices del grafo de control de flujo esta dado por un ordenamiento topológico de la transpuesta del grafo original. En un ordenamiento topológico se recorren cada uno de los vecinos de un nodo antes de continuar con el próximo nodo. Para explicar el algoritmo tomaremos como ejemplo el grafo de la Figura 2.17.a . En la Figura 2.17.b podemos observar la transpuesta del grafo original. Los posibles ordenamientos topológico para los nodos de este grafo son (4, 3, 2, 1) o (4, 2, 3, 1).

Dado que estamos trabajando con un grafo acíclico, dado un vértice cualquiera del grafo, puede suceder tres cosas:

- El vértice no tiene sucesores, este es el caso del nodo de salida.
- El vértice tiene dos sucesores, este es el caso en el que nos encontremos con un salto condicional.
- El vértice tiene solo un sucesor.

Podemos mapear esto directamente a GCL ya que un nodo con solo un sucesor corresponderá a una concatenación de instrucciones GCL y un nodo con dos sucesores corresponderá a una selección.

El algoritmo mantiene un diccionario M, en donde almacena la instrucción GCL correspondiente a cada vértice.



(a) Grafo de control de flujo original (b) Transpuesta del grafo

Figura 2.17: Grafo de control de flujo que se usara como ejemplo

Si el vértice no tiene sucesores entonces simplemente convertimos la instrucción del vértice a GCL y guardamos la instrucción en M. Podemos observar esto en la Figura 2.18.

Si el vértice tiene un sucesor, entonces la instrucción a guardar en M será una concatenación de GCL, la primera instrucción será la instrucción GCL del nodo sucesor y la segunda instrucción la conversión GCL del nodo actual. Notemos que como estamos recorriendo el grafo en orden topológico, siempre habremos visitado los sucesores de un nodo antes que el nodo mismo. Este caso lo podemos observar en las Figuras 2.19 y 2.20.

Finalmente si el vértice tiene dos sucesores, convertiremos la instrucción en una selección de GCL. La función branch_predicate nos devuelve la condición del salto, y la función split_suffix se encarga de devolvernos el postfijo que cada una de las instrucciones de los nodos sucesores tienen en común (gsuffix), la instrucción del primer sucesor quitando el postfijo (g1) y la instrucción del segundo sucesor quitándole el postfijo(g2). Ese ultimo caso podemos observarlo en la Figura 2.21.

Notemos que siempre existirá un postfijo para las instrucciones ya que el grafo es conexo y además tiene un nodo de entrada y un nodo de salida.

Una vez convertido el programa a GCL es posible realizar verificaciones sobre el programa mediante el uso de una variacion del algoritmo de weakest

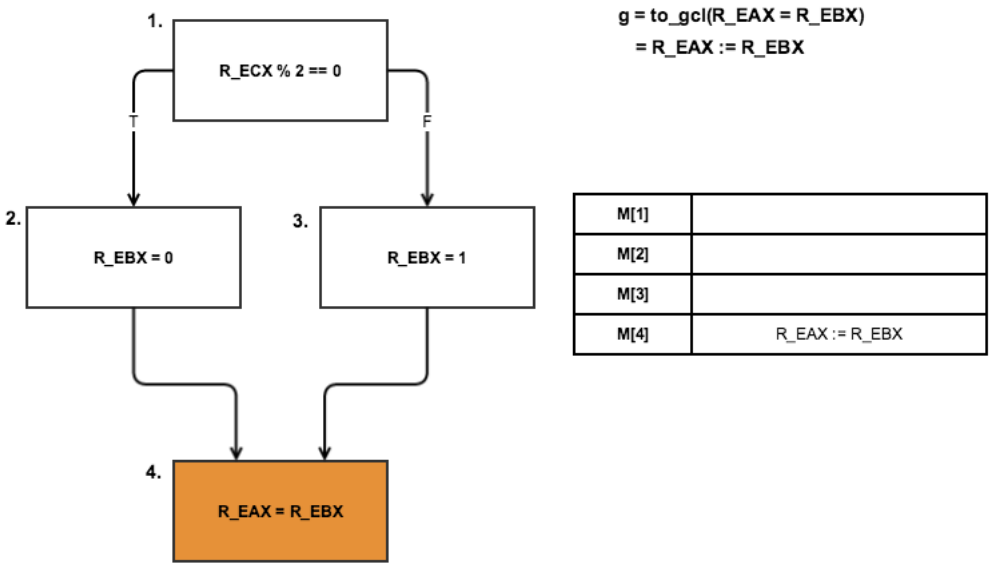


Figura 2.18: Corrida del algoritmo sobre el nodo 4. Tomamos el ordenamiento (4, 3, 2, 1)

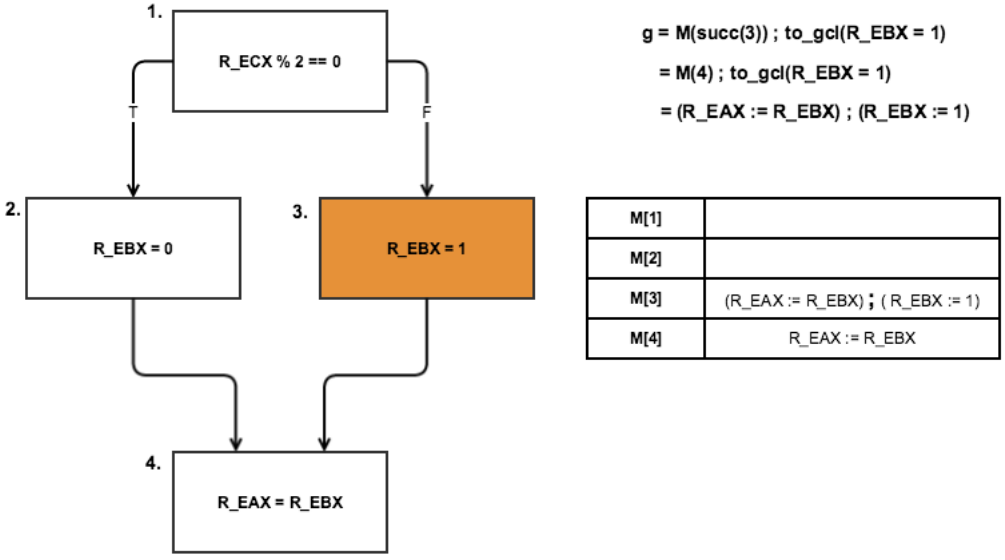


Figura 2.19: Corrida del algoritmo sobre el nodo 3.

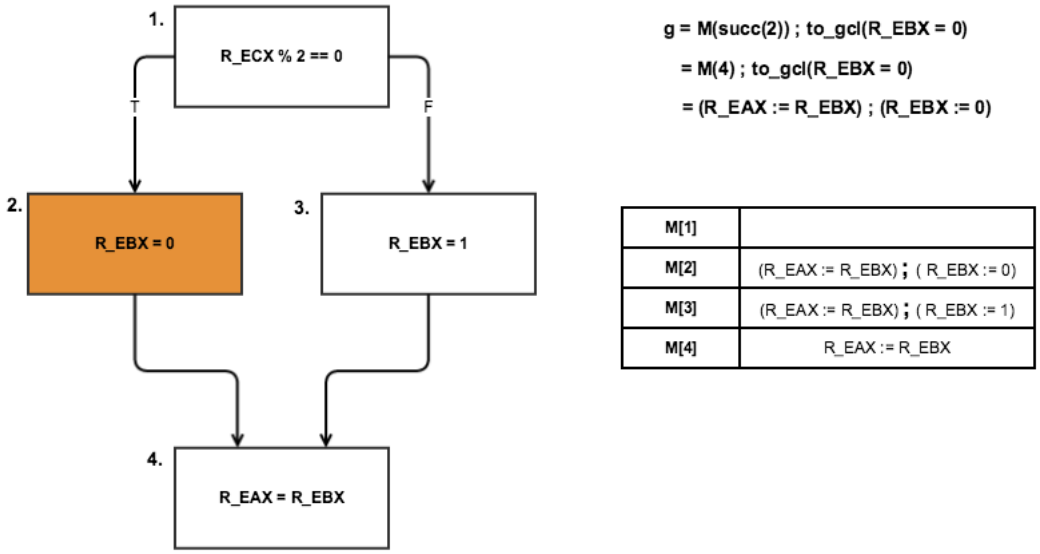


Figura 2.20: Corrida del algoritmo sobre el nodo 2.

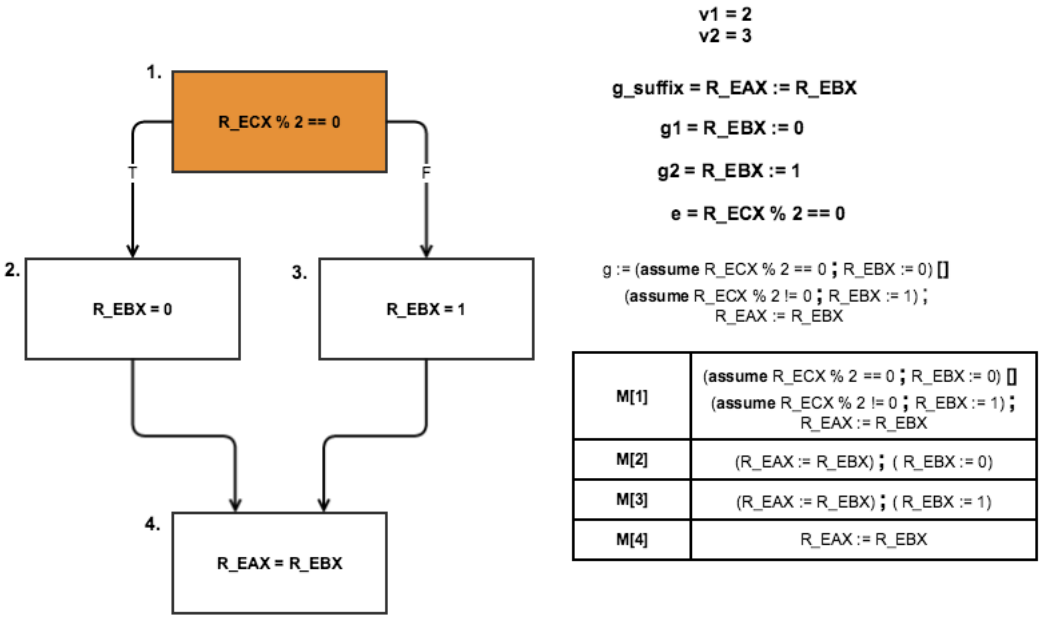


Figura 2.21: Corrida del algoritmo sobre el nodo 1.

precondition [13]. Nosotros utilizaremos esta utilidad para obtener formulas en lenguaje SMTLib2 para determinar su satisfacibilidad con el solver Z3.

Por ejemplo, para correr el algoritmo de precondición más débil (algoritmo por defecto) con postcondición EAX == 100, en el archivo de código intermedio codigo.il, y obtener la fórmula en el archivo codigo.fórmula (en formato SMTLibv2) deberíamos hacerlo así:

```
topredicate -il codigo.il -solver stp_smtlib -post 'R_EAX:
    u32 == 100:u32' -stp-out codigo.fórmula
```

2.4.3.4. ileval

Esta utilidad nos permite ejecutar un programa utilizando la semántica operacional de BAP [15]. Es de gran utilidad ya que nos permitirá "debugear" las modificaciones que hagamos sobre el lenguaje intermedio a la hora de realizar nuestros análisis.

Nos permite especificar un valor inicial para las variables no inicializadas, caso contrario, el programa determinará un valor para las mismas. El resultado de este programa es el valor de todas las variables utilizadas durante la ejecución del código intermedio.

2.4.3.5. codegen

Nos permite convertir de código intermedio de BAP a código intermedio LLVM [6]. El lenguaje intermedio de LLVM es ampliamente utilizado por la comunidad académica para realizar verificaciones, o transformaciones en programas. Generalmente es utilizado cuando se tiene el codigo fuente de la aplicación.

2.4.3.6. get_functions

Podemos utilizar esta utilidad para obtener un listado de las funciones de un código binario y además para convertir las funciones que queramos al código intermedio. La utilidad utiliza algunas heurísticas para determinar las funciones que posee un binario, por lo que es posible que no todas las funciones del binario aparezcan listadas. Para obtener resultados más confiables es recomendable aplicar la utilidad sobre programas que hayan sido compilados con símbolos de debug[2].

Para ver el listado de las funciones del programa, supongamos que queremos las funciones de test.exe, entonces deberíamos llamarlo de esta forma:

[2]http://en.wikipedia.org/wiki/Debug_symbol

```
get_functions test.exe
```

Luego, una vez que sabemos las funciones del programa, podemos extraer una función al código intermedio usando:

```
get_functions test.exe prefijo nombre_funcion
```

Esto producirá un archivo que tiene como nombre el prefijo concatenado con el nombre de la función y extensión il, que contendrá el código intermedio.

2.5. JSON

Habíamos dicho que una de las opciones de la utilidad toil era convertir el lenguaje intermedio a distintos formatos. Esto le da la posibilidad al programador de trabajar sobre el código intermedio utilizando un lenguaje de programación distinto de OCaml.

Uno de los formatos soportados, y el que nosotros utilizaremos, es JavaScript Object Notation (JSON). JSON es un formato estandarizado y abierto, soportado por la gran mayoría de los lenguajes de programación, utilizado para el intercambio de datos en aplicaciones web.

El formato soporta la representación de objetos, arreglos y valores.

2.5.1. Valores

Los valores básicos soportados son:

- Cadenas de caracteres unicode encerrados entre comillas.
- Los valores booleanos true y false.
- Un número.

A su vez, tenemos dos estructuras más complejas: listas y objetos, que también pueden ser utilizados como valores. A continuación veremos como se conforman estas estructuras:

2.5.2. Listas

Las listas son conjuntos de valores. Las listan comienzan con "[", terminan con "]", y tienen sus elementos separados por comas como podemos ver en la Figura 2.22

Por ejemplo, ["hola", 1, true] es una lista válida compuesta por una cadena de caracteres, un número y un booleano. Notar que una lista puede almacenar valores de distintos tipos.

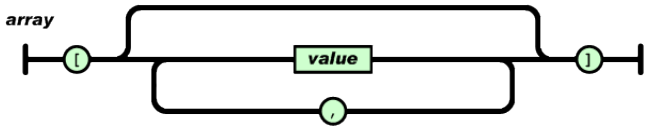


Figura 2.22: Lista JSON[5]

2.5.3. Objetos

Un objeto es un conjunto desordenado de pares nombre/valor. Los objetos comienzan con "{", terminan con "}", y sus elementos (pares nombre/valor) se encuentran separados por comas. Los elementos se representan mediante un nombre, que debe ser un string, y un valor. Un ejemplo válido de un objeto es {"números" : [1,2,3] , "texto" : "hola mundo"}. En este ejemplo tenemos dos pares nombre/valor, los nombres de cada par son "números" y "texto" respectivamente, el valor asociado a "números" es un arreglo de números y el valor de "texto" es una cadena de caracteres. En la Figura 2.23 podemos apreciar como se construye un objeto JSON.

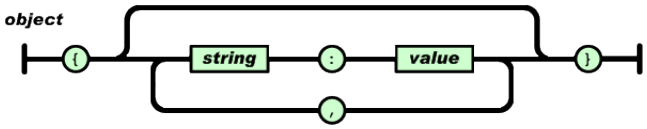


Figura 2.23: Objeto JSON[5]

2.5.4. Lenguaje intermedio usando JSON

Ahora veremos como se representa el lenguaje intermedio utilizando JSON.

Un programa en el lenguaje intermedio es representado como una lista compuesta de instrucciones BAP. Cada instrucción es representada mediante un objeto cuyo nombre hace referencia a la instrucción en cuestión, y cuyo valor es otro objeto con la información restante respecto de la instrucción. A continuación explicaremos como es el lenguaje intermedio para cada una de las instrucciones del lenguaje intermedio. Primero veremos como se representan los tipos, luego las expresiones, y finalmente las instrucciones.

2.5.4.1. Tipo (TYPE)

Estas son las representaciones de los tipos que vimos en 2.4.1.1.

- valores numéricos: un objeto con nombre "reg" y un número que indica la cantidad de bits.

  ```
  {'reg': cantidad_de_bits}
  ```

- arreglo: es un objeto con nombre "tmem" que tiene como valor otro objeto con nombre "index_type". Este objeto tiene a su vez como valor otro objeto que indica el tipo del arreglo.

  ```
  {'tmem': {'index_type': {u'reg': cantidad_de_bits}}}
  ```

2.5.4.2. Tipo BinOp (BINOP_TYPE)

Las operaciones binarias se diferencian entre si por su operador. Los operadores son cadenas de caracteres JSON. A continuación veremos una tabla con los operadores soportados.

Cadena de caracteres	Operador
plus	Adición
minus	Sustracción
times	Multiplicación
divide	División
sdivide	División con signo
modbop	módulo
smod	módulo con signo
lshift	Shift hacia la derecha
rshift	Shift hacia la izquierda
rshift	Shift hacia la izquierda con signo
andbop	And Lógico
orbop	Or Lógico
xor	Xor Lógico
eq	Igualdad
neq	Desigualdad
lt	Menor
le	Menor igual
slt	Menor con signo
sle	Menor igual con signo

2.5.4.3. Tipos UnOp (UNOP_TYPE)

Como en el caso de las expresiones binarias, las expresiones unarias también se diferencian por su operador y tienen como operador una cadena de caracteres JSON. Los operadores son:

Cadena de caracteres	Operador
neg	Negación bit a bit
not	Negación booleana

2.5.4.4. Tipos Cast (CAST_TYPE)

Lo mismo que para los tipos unarios y binarios vale para los tipos de casteo.

Cadena de caracteres	Operador
cast_unsigned	Incrementa la cantidad de bits sin conservar el signo.
cast_signed	Incrementa la cantidad de bits conservando el signo.
cast_high	Extrae los bits más altos de una expresión.
cast_low	Extrae los bits más bajos de una expresión.

2.5.4.5. Tipos Label (LABEL_TYPE)

El tipo label es un objeto JSON que puede ser de cualquier de las siguientes formas.

- `{'name' : Cadena de caracteres }`
- `{'addr' : Entero }`

2.5.4.6. Variables (VAR)

Las variables del lenguaje intermedio (Sección 2.4.1.2) se representan mediante objetos JSON. El nombre del objeto es "var" y su valor es otro objeto que tiene 3 pares nombre/valor.

- "id" es un número que sirve para identificar a la variable.
- "name" es un string que da nombre a la variable.
- "typ" un tipo básico.

Como ejemplo tomemos la representación JSON de la variable T_t1 de la línea 3 de la Figura 2.11.

```
{'var': {'id': 1,
         'name': 'T_t1',
         'typ': {'reg': 32}
        }
}
```

2.5.4.7. Expresiones (EXP)

Int La expresión para representar números enteros es un objeto con nombre "inte" que tiene como valor otro objeto compuesto de 2 pares nombre/valor.

- "int": es una cadena de caracteres que representa al número en cuestión.
- "typ": es uno de los tipos numéricos definidos en 2.5.4.1.

```
{'inte': {'int' : String representing the integer,
          'typ' : TYPE
          }
}
```

Por ejemplo, la representación del número 2 que se encuentra en la parte derecha de la asignación en la línea 4 de la Figura 2.11 es:

```
{'inte': {'int' : '2',
          'typ' : {'reg': 32}
          }
}
```

BinOp La expresión para representar operaciones binarias es un objeto con nombre "binop" que tiene como valor otro objeto compuesto de 3 pares nombre/valor.

- "binop_type": es uno de los tipos definidos en 2.5.4.2.
- "l_exp": es una expresión que representa uno de los operandos de la operación.
- "r_exp": es otra expresión que representa el otro operando de la operación.

```
{'binop': {'binop_type' : BINOP_TYPE,
                  'l_exp'       : EXP,
                  'r_exp'       : EXP,
          }
}
```

Tomemos como ejemplo la adición de la parte derecha de la asignación de la línea 5 de la Figura 2.11.

```
{'binop': {
     'binop_type' : 'plus',
     'l_exp' : {'var': {'id': 1  ,
                        'name' : 'R_EAX',
                        'typ' : {'reg': 32}
                        }
                },
     'r_exp' : {'var': {'id': 2  ,
                         'name' : 'T_t2',
                         'typ' : {'reg': 32}
                        }
```

```
                    },
    }
}
```

UnOp La expresión para representar operaciones unarias es un objeto con nombre "unop" que tiene como valor otro objeto compuesto de 2 pares nombre/valor.

- "unop_type": es uno de los tipos definidos en 2.5.4.3.
- "exp": es la expresión sobre la que se aplicar el operando.

```
{'unop': {
    'unop_type': UNOP_TYPE,
    'exp'      : EXP
    }
}
```

Podemos ver como ejemplo la negación bit a bit de la línea 7 en la Figura 2.11.

```
{'unop': {
    'unop_type': 'neg',
    'exp'      : {'var':
                        {'id': 1 ,
                         'name' : 'T_t2',
                         'typ' : {'reg': 32}
                        }
                 }
    }
}
```

Var Es la misma que fue definida anteriormente en 2.5.4.6.

Cast La expresión para representar casteos de tipo es un objeto con nombre "cast" que tiene como valor otro objeto compuesto de 3 pares nombre/valor.

- "cast_typ": es uno de los tipos definidos en 2.5.4.4.
- "new_typ": es uno de los tipos definidos en 2.5.4.1.
- "exp": es la expresión sobre la que ejecutaremos el casteo.

```
{'cast': {
    'cast_type' : CAST_TYPE,
    'new_type'  : TYPE,
    'exp'       : EXP
    }
}
```

Tomemos como ejemplo el casteo de tipo high de la parte derecha de la asignación de la línea 13 de la Figura 2.11. La representación es:

```
{'cast': {
    'cast_type' : 'cast_high',
    'new_type'  : 'typ' : {'reg': 1},
    'exp'       : {'var':
                        {'id': 2 ,
                         'name' : 'R_EAX',
                         'typ' : {'reg': 32}
                        }
                  }
    }
}
```

Load La expresión para leer datos de un arreglo es representada mediante un objeto con nombre "load" que tiene como valor otro objeto compuesto de 4 pares nombre/valor.

- "address": es una expresión que contendrá la dirección de memoria desde la cual se leerán los datos.
- "endian": es una expresión que determina el tipo de endianes de la expresión. El endianess es representado por BAP como una variable booleana, 0 representa little endian, mientras que 1 representa big endian.
- "memory": es una expresión que representa la variable de tipo arreglo desde la cual se leerán los datos.
- "typ": es un tipo de los tipos definidos en 2.5.4.1 que indica cuantos bytes se leerán.

```
{'load': {
    'address' : EXP,
    'endian'  : EXP,
    'memory'  : EXP,
```

```
    'typ'       : TYPE
    }
}
```

Podemos ver como ejemplo la lectura de datos en la parte derecha de la línea 3 en la Figura 2.12.

```
{'load': {
    'address' : {'inte': {'int' : '4243936', 'typ' : {'reg': 32}}},
    'endian'  : {'inte': {'int' : '0', 'typ' : {'reg': 1}}},
    'memory'  : {'var': {'id':56 ,
                         'name':'mem',
                         'typ':{'tmem':{'index_type':{'reg':32}}}}}
                }
    'typ'     : 'typ' : {'reg': 32}
    }
}
```

Store La expresión para escribir datos en una variable de tipo arreglo es representada mediante un objeto con nombre "store" que tiene como valor otro objeto compuesto de 5 pares nombre/valor.

- "address": es una expresión que contendrá la dirección de memoria en la cual se escribirán los datos.
- "endian": es una expresión que determina el tipo de endianes de la expresión. El endianess es representado por BAP como una variable booleana. 0 representa little endian, mientras que 1 representa big endian.
- "memory": es una expresión que representa la variable de tipo arreglo en la cual se escribirán los datos.
- "typ": es un tipo de los tipos definidos en 2.5.4.1 que indica cuantos bytes se escribirán.
- "value": es una expresión que contendrá el valor a almacenar.

```
{'store': {
    'address' : EXP,
    'endian'  : EXP,
    'memory'  : EXP,
    'typ'     : TYPE,
    'value'   : EXP
    }
}
```

Tomemos como ejemplo la escritura de la variable de tipo arreglo de la parte derecha de la asignación de la línea 6 en la Figura 2.12. La representación es:


```
{'store': {
    'address' : "binop":{
        "binop_type":"plus",
        "lexp":{"var":{"name":"R_EBP","id":0,"typ":{"reg":32}}},
        "rexp":{"inte":{"int":"4294967256","typ":{"reg":32}}}}
        },
    'endian' : {"inte":{"int":"0","typ":{"reg":1}}},
    'memory' : {"var":{"name":"mem",
                "id":56,
                "typ":{"tmem":{"index_type":{"reg":32}}}}
                },
    'typ'    : {"typ":{"reg":32}},
    'value'  : {"inte":{"int":"0","typ":{"reg":32}}}
    }
}
```


2.5.4.8. Instrucciones

Atributos (ATTR) Todas las instrucciones están compuestas por atributos, esto proporcionan información adicional sobre la instrucción en cuestión. Algunos de los posibles atributos son:

- Asm: representación en ensamblador de la instrucción.
- Address: la dirección (dentro del binario) de la instrucción.
- Liveout: indica que esta instrucción debe ser considerada como "viva" para los algoritmos de eliminación de código muerto.
- Synthetic: indica que la instrucción fue agregada por algún algoritmo de análisis.
- StrAttr: es un string genérico.

La representación de los atributos en JSON se realiza mediante una lista de objetos JSON, en donde cada uno de los objetos de esa lista representa un atributo. Por ejemplo, los atributos de la línea 1 de la Figura 2.12 se representan de esta manera:


```
[{"asm":"mov     0x40c1e0,%eax"}]
```


Como la instrucción solo tiene un atributo, la representación es una lista con solo un elemento.

Asignación La representación de una asignación es un objeto con nombre 'move' cuyo valor es otro objeto compuesto por 3 pares nombre/valor:

- 'var': es una variable como la definida anteriormente, indica donde se guardaran los datos.
- 'exp': es una expresión que indica el valor que será asignado a la variable.
- 'attributes': es una lista de atributos definidos en 2.5.4.8.

```
{'move': {
          'var'          : VAR,
          'exp'          : EXP,
          'attributes'   : ATTR,
         }
}
```

Tomemos como ejemplo la escritura de la variable de tipo arreglo de la parte derecha de la asignación de la línea 3 en la Figura 2.11. La representación es:

```
{'move': {
     'var'          : {"name":"T_t1","id":78,"typ":{"reg":32}},
     'exp'          : {"var":
                          {"name":"R_EAX",
                           "id":5,"typ":{"reg":32}
                          }
                      },
     'attributes'   : [],
     }
}
```

Salto condicional Un salto condicional es un objeto con nombre 'cjmp' cuyo valor es otro objeto compuesto por 4 pares nombre/valor:

- 'cond': es una expresión booleana que determinará hacia donde sigue la ejecución del programa.
- 'iftrue': es la expresión que se ejecutará en caso de que la condición sea verdadera.
- 'iffalse': es la expresión que se ejecutará en caso de que la condición sea falsa.

- 'attributes': es una lista de atributos definidos en 2.5.4.8.

```
{'cjmp': {
        'cond'       : EXP,
        'iftrue'     : EXP,
        'iffalse'    : EXP,
        'attributes' : ATTR,
        }
}
```

Salto incondicional Un salto incondicional es un objeto con nombre 'jmp' cuyo valor es otro objeto compuesto por 2 pares nombre/valor:

- 'exp': es la expresión que indicara hacia donde se producirá el salto.
- 'attributes': es una lista de atributos definidos en 2.5.4.8.

```
{'jmp': {
        'exp'        : EXP,
        'attributes' : ATTR
        }
}
```

Label Un label es un objeto con nombre 'label' cuyo valor es otro objeto compuesto por 2 pares nombre/valor:

- label: es un tipo definido en 2.5.4.5.
- 'attributes': es una lista de atributos definidos en 2.5.4.8.

```
{'label': {
        'label'      : LABEL_TYPE,
        'attributes' : ATTR
        }
}
```

Tomemos como ejemplo el label de la línea 1 de la Figura 2.12.

```
{'label': {
        'label'      : {"addr":0x401046},
        'attributes' : {"asm":"mov    0x40c1e0,%eax"}
        }
}
```

Halt Halt es un objeto con nombre 'halt' cuyo valor es otro objeto compuesto por 2 pares nombre/valor:

- exp: es una expresión que indica el valor que devolverá el programa una vez que termine la ejecución.
- 'attributes': es una lista de atributos definidos en 2.5.4.8.

```
{'halt': {
        'exp'          : EXP,
        'attributes'   : ATTR
                }
}
```

Assert Assert es un objeto con nombre 'assert' cuyo valor es otro objeto compuesto por 2 pares nombre/valor:

- exp: es una expresión que se evalúa y en caso de ser falsa detiene la ejecución del programa.
- 'attributes': es una lista de atributos definidos en 2.5.4.8.

```
{'assert_stmt': {
        'exp'          : EXP,
        'attributes'   : ATTR
        }
}
```

Special Special es un objeto con nombre 'special' cuyo valor es otro objeto compuesto por 2 pares nombre/valor:

- string: es el nombre del procedimiento definido externamente que se esta tratando de llamar.
- 'attributes': es una lista de atributos definidos en 2.5.4.8.

```
{'special': {
        'string' : Cadena de caracteres,
        'attributes'   : ATTR
        }
}
```

Comentario Un comentario es un objeto con nombre 'comment' cuyo valor es un string con el comentario propiamente dicho.

- string: es una cadena de caracteres con el comentario propiamente dicho.
- 'attributes': es una lista de atributos definidos en 2.5.4.8.

```
{'comment': {
        'string'       : string,
        'attributes'   : ATTR
                }
}
```

Capítulo 3

Método Desarrollado

En esta parte explicaremos el método desarrollado que nos permitirá detectar la presencia de buffer overflow en código binario utilizando BAP.

El método desarrollado consiste, a grandes rasgos, en:

1. Extracción del código intermedio del binario en el que se desee detectar la presencia de buffer overflow mediante la utilidad toil.

2. Exportación del código intermedio obtenido en el punto 1 a JSON mediante la utilidad toil.

3. Parseo del código intermedio (JSON) y conversión de las instrucciones a su representación en clases de Python (código intermedio Python).

4. Modificación del código intermedio (Python) para poder detectar buffer overflow y creación de una postcondición que permitirá la verificación.

5. Conversión del código intermedio (Python) a código intermedio de BAP.

6. Chequeo de buffer overflow mediante la utilidad topredicate y la postcondición obtenida en el punto 4.

Para llevar acabo esto fue necesario desarrollar los siguientes módulos:

- Parser del lenguaje intermedio de BAP en formato JSON (Sección 3.2). Este módulo toma el lenguaje intermedio exportado en formato JSON y convierte cada una de las instrucciones y expresiones a su equivalente en Python (paso 3). El nombre del módulo desarrollado es parser.

- Módulos para representar las instrucciones y expresiones del lenguaje intermedio en Python (Sección 3.1). Para cada una de las instrucciones

y expresiones del lenguaje intermedio se creo una clase de Python que las representa (paso 3). Este módulo se encarga también de la conversión de la representación al lenguaje intermedio nuevamente (paso 5). El nombre del módulo desarrollado es bap_il.

- módulo para modificar la semántica de las instrucciones para el chequeo de buffer overflow (Sección 3.5). Este módulo modifica el código intermedio Python agregando los elementos necesarios (paso 4) para que después pueda ser posible el chequeo mediante la utilidad topredicate (paso 6). El nombre de este módulo es check_buffer_overflow.py.

En la Figura 3.1 podemos observar la relación entre los módulos desarrollados, las utilidades de BAP y los pasos mencionados anteriormente de manera gráfica. Notemos que en la parte superior se encuentran las utilidades que provee BAP mientras que en la parte inferior se encuentran los módulos desarrollados.

Al tratar de agregar el soporte para esta verificación surgieron problemas que no hacen a la verificación en si, a saber:

- BAP no soporta análisis interprocedurales.
- La gran mayoría de los binarios tienen funciones importadas, cuyo código reside fuera del binario en cuestión.

La solución al primer problema fue realizar el inlining de las funciones llamadas, es decir, remplazar cada llamado a una función (no importada) por el cuerpo de la misma, en la sección 3.3 se explicará este proceso en más detalle.

Para el segundo problema, se remplazaron las funciones importadas por equivalentes semánticos, esto se explicará más claramente en la sección 3.4.

Si bien el código desarrollado para solucionar ambos problemas se encuentra en check_buffer_overflow.py, se explicarán estos dos ítem en secciones separadas debido a que, como dijimos anteriormente, no hacen a la verificación de buffer overflow en si.

En la Figura 3.2 podemos ver un diagrama de los módulos desarrollados. Como podemos ver, check_buffer_overflow.py hace uso de los módulos bap_il y parser. A su vez, cada uno de estos módulos esta compuesto de submódulos para el manejo de las instrucciones, expresiones, tipos y atributos.

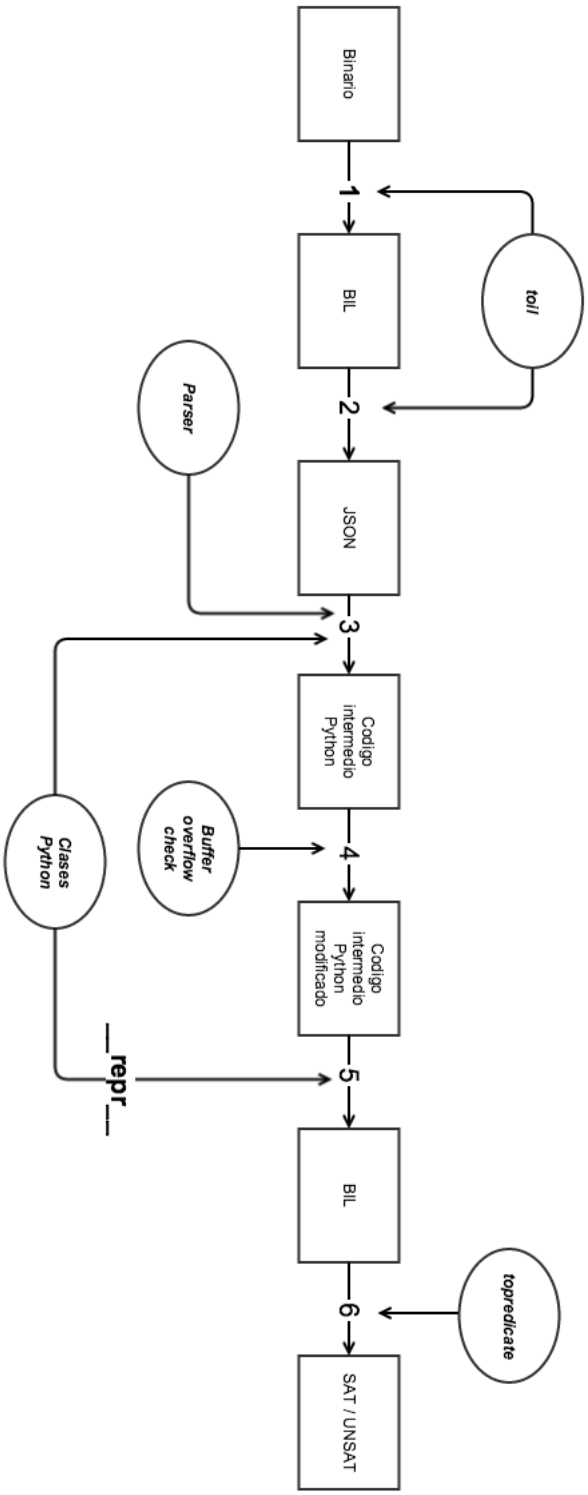


Figura 3.1: Diagrama del método desarrollado.

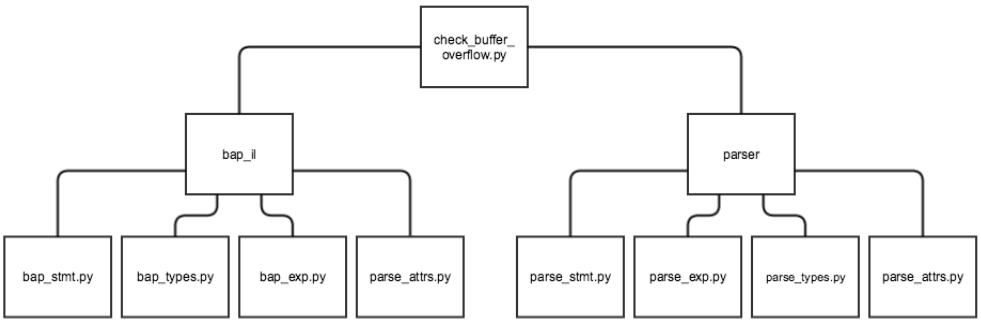


Figura 3.2: Diagrama de módulos

3.1. Representación en Python

Para cada uno de los atributos, tipos, instrucciones y expresiones de BAP se definió una clase de python para representarlas. Cada una de estas clases encapsula toda la información necesaria para representar un atributo, tipo, expresión o una instrucción.

Todas las clases siguen un patrón de diseño similar:

- Los atributos de la clase son las expresiones por las que esta compuesta la instrucción y sus atributos.

- Todas las clases tienen implementada la función __repr__.

La función __repr__ nos permite obtener una representación de una instancia de un objeto en Python. En nuestro caso, la representación elegida es la sintaxis del lenguaje intermedio de BAP para cada instrucción, expresión, tipo o atributo. Esto quedará más claro una vez que el lector vea las representaciones de las clases desarrolladas a lo largo de la sección y compare la representación de cada objeto con el lenguaje intermedio de BAP de la sección 2.4.1.

3.1.1. Atributos

Para representar los atributos se definió la clase genérica Attr que agrupa a todas las clases de atributos y, para cada uno de los atributos se definieron las siguientes clases:

Nombre de la clase	Atributo
Asm	String con el código asm de la instrucción
Address	Dirección de una instrucción
StrAttr	String con un valor arbitrario
Liveout	Usado en la eliminación de código muerto
Thread	número de thread
Synthetic	Instrucción agregada por un algoritmo de análisis
Context	Usado en las trazas de la pintool.

En la Figura 3.3 podemos ver la clase que representa al atributo Asm.

```
class Asm(Attr):
    def __init__(self, asm=None):
        self.asm = asm
    def __repr__(self):
        return '@asm "{0}"'.format(self.asm)
```

Figura 3.3: Clase que representa al atributo Asm.

3.1.2. Tipos

Para representar los tipos de BAP se definieron dos clases genéricas: BasicType y LabelType.

BasicType es una clase genérica que será el padre de las dos clases que usaremos para representar los tipos básicos de BAP, el tipo de los valores numéricos y el tipo de los arreglos.

El tipo de los valores numéricos será representado mediante la clase Register que podemos ver en la Figura 3.4.

```
class Register(BasicType):
    def __init__(self, bits=None):
        self.bits = bits
    def __repr__(self):
        if self.bits == 1:
            return 'bool'
        else:
            return 'u{0}'.format(str(self.bits))
```

Figura 3.4: Clase que representa a los valores numéricos

Mientras que los arreglos son representados mediante la clase TMem, que podemos ver en la Figura 3.5.

```
class TMem(BasicType):
    def __init__(self, index_type=None):
        self.index_type = index_type
    def __repr__(self):
        return '?u{0}'.format(str(self.index_type))
```

Figura 3.5: Clase que representa a los arreglos.

Los dos tipos de labels que se definieron fueron StrLabel y AddrLabel, podemos ver su representación en las Figuras 3.6 y 3.7.

```
class StrLabel(LabelType):
    def __init__(self, name=None):
        self.name = name
    def __repr__(self):
        return 'label {0}'.format(str(self.name))
```

Figura 3.6: Clase que representa los label de tipo cadena de caracteres.

```
class AddrLabel(LabelType):
    def __init__(self, addr=None):
        self.addr = addr
    def __repr__(self):
        return 'addr 0x%x'% self.addr
```

Figura 3.7: Clase que representa los label de tipo numérico.

3.1.3. Expresiones

Para las expresiones se definió una clase genérica Exp que será la clase padre de todas las expresiones. Para la representación de cada una de las expresiones de la sección 2.4.1.2 se implementaron las siguientes clases que podemos observar en el Cuadro 3.1.

En la Figura 3.8 podemos ver la representación de una expresión binaria.

Nombre de la clase	Instrucción
Variable	Variable
Int	Constante numérica
BinOp	Operador binario
Unop	Operador unario
Cast	Casteo
Load	Load
Store	Store

Cuadro 3.1: Clases para representar las expresiones.

```
1  class BinOp(Exp):
2      def __init__(self, binop_type=None, lexp=None, rexp=None):
3          """
4          @binop_type: binop_types
5          @lexp: Exp
6          @rexp: Exp
7          """
8          self.binop_type = binop_type
9          self.lexp = lexp
10         self.rexp = rexp
11
12     def __repr__(self):
13         if type(self.lexp) in [var_type, int_type]:
14             final_lexp = '{0}'.format(self.lexp)
15         else:
16             final_lexp = '({0})'.format(self.lexp)
17             if type(self.rexp) in [var_type, int_type]:
18                 final_rexp = '{0}'.format(self.rexp)
19             else:
20                 final_rexp = '({0})'.format(self.rexp)
21         return '{0} {1} {2}'.format(final_lexp,
22                                     binop_types[self.binop_type],
23                                     final_rexp)
```

Figura 3.8: Clase que representa una operación binaria.

Para representar cada uno de los operadores, tanto binarios, unarios y de casteo se utilizaron diccionarios de Python. Estos tienen como clave la representación JSON del operador y como valor el símbolo o cadena de caracteres del lenguaje intermedio correspondiente a ese operador.

En la Figura 3.9 podemos ver el diccionario para los operadores binarios. Si observamos las tablas de los operadores JSON (sección 2.5.4.2) y la tabla de la sintaxis de las expresiones binarias (sección 2.4.1.2) veremos que existe un mapeo directo.

Podemos observar en la línea 22 del código de la Figura 3.8 como se utiliza el diccionario binop_types para determinar el operador de la expresión.

```
binop_types = {}
binop_types['plus'] = '+'
binop_types['minus'] = '-'
binop_types['times'] = '*'
binop_types['divide'] = '/'
binop_types['sdivide'] = '$/'
binop_types['mod'] = '%'
binop_types['smod'] = '$%'
binop_types['lshift'] = '<<'
binop_types['rshift'] = '>>'
binop_types['arshift'] = '$>>'
binop_types['andbop'] = '&'
binop_types['orbop'] = '|'
binop_types['xor'] = '^'
binop_types['eq'] = '=='
binop_types['neq'] = '<>'
binop_types['lt'] = '<'
binop_types['le'] = '<='
binop_types['slt'] = '$<'
binop_types['sle'] = '$<='
```

Figura 3.9: Diccionario para los operadores binarios.

3.1.4. Instrucciones

Para las instrucciones se definió una clase genérica Statement que será la clase padre de todas las instrucciones. Para la representación de cada una de las instrucciones de la sección 2.4.1.3 se implementaron las clases que podemos observar en el cuadro 3.2.

Nombre de la clase	Instrucción
Move	Asignación
Jmp	Salto incondicional
CJmp	Salto condicional
Label	Label
Halt	Halt
Special	Special
Assert	Assert
Comment	Comentario

Cuadro 3.2: Clases para representar las instrucciones.

En la Figura 3.10 podemos ver la clase que representa un salto indirecto, si vemos la representación JSON de la instrucción en la sección 3.1.4 podremos ver que los atributos de la clase son los mismos que componen al objeto JSON.

```
class CJmp(Statement):
  def __init__(self,  cond = None,
               iftrue=None, iffalse=None,
               attrs=None):
    """
    @cond: Exp
    @iftrue: Exp
    @iffalse: Exp
    @attrs: List of atributes get by parse_attrs
    """

    self.cond = cond
    self.iftrue = iftrue
    self.iffalse = iffalse
    self.attrs = attrs

  def __repr__(self):
    if self.attrs:
      return 'cjmp {0}, {1}, {2} {3}'.format(self.cond,
                                             self.iftrue,
                                             self.iffalse,
                                             self.attrs)
    else:
      return 'cjmp {0}, {1}, {2}'.format(self.cond,
                                         self.iftrue,
                                         self.iffalse)
```

Figura 3.10: Clase para representar un salto indirecto

3.2. Parser

Para la realización del parser, se tuvo que tener en cuenta el módulo que se encarga de la exportación del lenguaje intermedio en formato JSON del código fuente de BAP, ya que en este módulo se encuentra detallado como se construyen los objetos JSON para cada una de las instrucciones y expresiones de BAP.

3.2.1. Atributos (parse_attrs.py)

Para cada uno de los atributos del lenguaje se definió una función de parseo particular. Por ejemplo, en la Figura 3.11 podemos ver el código que parsea un atributo de tipo Asm. Notemos que las funciones de parseo devuelven un objeto Python de uno de los tipos definidos en la sección 3.1.1.

```
def parse_asm_attr(data):
    return Asm(data['asm'])
```

Figura 3.11: Subfunción de parseo de un atributo de tipo Asm.

Como los atributos de una instrucción son una lista de objetos JSON, es necesario llamar a la función de parseo definida para cada atributo sobre cada uno de los objetos de la lista. Para llevar a cabo esto, se definió la función de la Figura 3.12.

```
def parse_attrs(data):
    parse_attr = lambda x: attributes_parse_functions[x.keys()[0]](x)
    return AttrList(map(parse_attr,data))
```

Figura 3.12: Función principal del parseo de atributos.

Esta función se vale de un diccionario de Python que tiene como clave un string con el nombre de un atributo, y como valor la función encargada de parsearlo. Este diccionario actúa como un dispatcher de las funciones de parseo para cada uno de los atributos. Podemos ver el diccionario en la Figura 3.13. Notemos que las llaves del diccionario se corresponden a los nombres de los objetos JSON que representan a los atributos.

3.2.2. Tipos (parse_types.py)

Para parsear los tipos del lenguaje se utilizaron dos funciones: una para los tipos básicos (numérico y arreglo) y otro para los tipos label.

```
attributes_parse_functions = {}
attributes_parse_functions['asm'] = parse_asm_attr
attributes_parse_functions['address'] = parse_address_attr
attributes_parse_functions['strattr'] = parse_str_attr
attributes_parse_functions['thread_id'] = parse_thread_attr
attributes_parse_functions['liveout'] = parse_liveout_attr
attributes_parse_functions['context'] = parse_context_attr
attributes_parse_functions['synthetic'] = parse_synthetic_attr
```

Figura 3.13: Diccionario del dispatcher de parseo de atributos.

La función para parsear los tipos básicos se vale del nombre del objeto JSON para determinar si se trata de una variable de tipo numérico, o de una variable de tipo arreglo. Recordemos, de la sección 2.5.4.1, que el objeto JSON que representa a una variable de tipo numérico tiene como nombre al string 'reg', mientras que si se tratara de una variable de tipo arreglo el nombre sería el string 'tmem'. En la Figura 3.14 podemos observar el código del parser de tipos básicos.

```
def parse_basic_type(data):
    if data.keys()[0] == 'reg':
        return Register(data['reg'])
    else:
        return TMem(data['tmem']['index_type']['reg'])
```

Figura 3.14: Función de parseo de los tipos básicos

Para parsear un label seguimos una lógica similar a la utilizada para los tipos básicos. Si recordamos de la sección 2.5.4.5, los objetos JSON que representan a los labels también se diferencian entre si por el nombre del par nombre/valor. En la Figura 3.15 podemos ver la función de parseo de los labels.

```
def parse_label_type(data):
    if data.keys()[0] == 'name':
        return StrLabel(data['name'])
    else:
        return AddrLabel(data['addr'])
```

Figura 3.15: Función de parseo de labels

En ambos casos, notemos que las funciones de parseo devuelven un objeto Python de uno de los tipos definidos en la sección 3.1.2.

3.2.3. Instrucciones (parse_stmt.py)

El código del parser de instrucciones consiste básicamente, al igual que el parser de atributos, en identificar el tipo de instrucción que estamos parseando y llamar a la función encargada de parsear ese tipo particular de instrucción.

La función principal del parser es parse_statement, podemos ver el código en la Figura 3.16. Si recordamos la estructura JSON de las instrucciones, estas están compuestos por un par nombre/valor donde el nombre es el tipo de instrucción y el valor es otro objeto que contiene los datos de la instrucción. Esta función toma el nombre del objeto JSON que representa la instrucción y dependiendo del tipo de instrucción llama a la función correspondiente.

```
def parse_statement(json_stmt):
    stmt = json_stmt.keys()[0]
    data = json_stmt[stmt]
    statement = statement_parse_functions[stmt](data)
    return statement
```

Figura 3.16: Función principal de parseo de instrucciones

statement_parse_functions es un diccionario con funciones de parseo para cada una de las instrucciones de BAP. Esta definido como podemos ver en la Figura 3.17

```
statement_parse_functions = {}
statement_parse_functions['move'] = parse_stmt_move
statement_parse_functions['cjmp'] = parse_stmt_cjmp
statement_parse_functions['jmp'] = parse_stmt_jmp
statement_parse_functions['label_stmt'] = parse_stmt_label
statement_parse_functions['halt'] = parse_stmt_halt
statement_parse_functions['assert_stmt'] = parse_stmt_assert
statement_parse_functions['comment'] = parse_stmt_comment
statement_parse_functions['special'] = parse_stmt_special
```

Figura 3.17: Diccionario del dispatcher de parseo de instrucciones

Finalmente, cada una de las funciones de parseo de instrucciones, toma los datos de la instrucción que le paso el dispatcher principal y, dependiendo

de como este compuesta la instrucción, llama a subfunciones de parseo para las expresiones y atributos que la componen como podemos ver en la Figura 3.18. Notemos que las funciones de parseo devuelven un objeto Python de uno de los tipos definidos en la sección 3.1.4.

```
def parse_stmt_cjmp(data):
    cond = parse_expression(data['cond'])
    attrs = parse_attrs(data['attributes'])
    iffalse = parse_expression(data['iffalse'])
    iftrue = parse_expression(data['iftrue'])
    return CJmp(cond, iftrue, iffalse, attrs)
```

Figura 3.18: Subfunción de parseo de un salto condicional

3.2.4. Expresiones (parse_expr.py)

El parseo de la expresiones y atributos sigue la misma lógica que el parseo de instrucciones, es decir, un dispatcher general de expresiones y atributos que llama a subfunciones para parsear cada uno de los distintos tipos de expresiones de BAP.

En la Figura 3.19 podemos ver la función principal de parseo de las expresiones.

```
def parse_expression(json_exp):
    exp = json_exp.keys()[0]
    data = json_exp[exp]
    expression = expression_parse_functions[exp](data)
    return expression
```

Figura 3.19: Función principal del parseo de expresiones

La función de parseo se vale de un diccionario que tiene definida una función de parseo particular para cada una de las funciones. En la Figura 3.20 podemos observar ese diccionario.

Finalmente, en la Figura 3.21, podemos observar la función de parseo de una expresión binaria. Notemos como cada función de parseo devuelve un objeto de python que representa a la expresión parseada.

```
expression_parse_functions = {}
expression_parse_functions['var'] = parse_var
expression_parse_functions['load'] = parse_load
expression_parse_functions['store'] = parse_store
expression_parse_functions['binop'] = parse_binop
expression_parse_functions['unop'] = parse_unop
expression_parse_functions['lab'] = parse_lab
expression_parse_functions['inte'] = parse_int
expression_parse_functions['cast'] = parse_cast
expression_parse_functions['ite'] = parse_ite
expression_parse_functions['extract'] = parse_extract
expression_parse_functions['concat'] = parse_concat
expression_parse_functions['let'] = parse_let
expression_parse_functions['unknown'] = parse_unknown
```

Figura 3.20: Diccionario del dispatcher de parseo de expresiones

```
def parse_binop(data):
    lexp = parse_expression(data['lexp'])
    rexp = parse_expression(data['rexp'])
    binop_type = data['binop_type']
    return BinOp(binop_type, lexp, rexp)
```

Figura 3.21: Subfunción de parseo de una expresión binaria.

3.3. Análisis interprocedural

Ahora veremos porque BAP no soporta análisis formales en los que se realice llamado entre funciones. Esto se debe a que no existe soporte para saltos indirectos, como el introducido por la instrucción RET. Decimos que el salto es indirecto debido a su naturaleza dinámica. El salto se realiza hacia una dirección que esta almacenada en memoria, y no hay forma a priori de saber ese valor de memoria.

Si recordamos, en la sección 2.2.2.2 explicamos que cuando se ejecuta una instrucción CALL, se pushea en el stack la dirección de la próxima instrucción a ejecutar. La función llamada termina con una instrucción RET que toma el valor del stack que fue pusheado por CALL y salta a la función que la llamo originalmente para así seguir con el curso de la ejecución del programa.

La forma más sencilla de agregar el soporte para análisis interprocedurales es el inlining. Esta técnica consiste en remplazar cada ocurrencia de un CALL con el cuerpo de la función a la que estamos llamando.

En un primera instancia se intentó modificar directamente el código binario para realizar el inlining. Para entender las complicaciones de este método, primero es necesario saber que en los sistemas operativos modernos es común que el mismo binario se cargue en distintas direcciones de memoria durante distintas ejecuciones. Es por ello que los compiladores crean binarios que son re-localizables, es decir, funcionan independientemente de la dirección en la que sean cargados en memoria. Una de las formas en las que esto se lleva a cabo es mediante el uso de saltos relativos. Estos son saltos hacia adelante o atrás de una determinada cantidad de bytes, pero sin especificar una dirección absoluta de memoria.

Al remplazar una instrucción de CALL por el cuerpo de la función, estamos agregando bytes al binario, lo que produce que algunos de los saltos relativos del programa apunten hacia direcciones incorrectas, cambiando así la semántica del programa.

Además, al agregar más bytes, es posible que se desfasen algunas secciones del programa (.data, .rdata, etc.) y, en caso de necesitarse agregar más bytes en la sección .text de los que tenemos disponibles, también se hace necesario modificar el tamaño de la sección .text en los headers del ejecutable.

Para evitar estos problemas se decidió realizar el inlining directamente en el código intermedio generado por BAP. Explicaremos el funcionamiento del método mediante ejemplos y luego mostraremos el código desarrollado. El primer ejemplo que utilizaremos es el de la Figura 3.22.

Como podemos observar, este código realiza un llamado a la función resta. En la Figura 3.23 podemos observar el código ensamblador generado para este programa.

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

int resta(int a, int b){
    return a - b;
}

int main(int argc, char *argv[]) {
    int num;
    int result = resta(5,5);
    return result;
}
```

Figura 3.22: Programa con un solo llamado a función

```
080483dc <resta>:
 80483dc:       55                      push   %ebp
 80483dd:       89 e5                   mov    %esp,%ebp
 80483df:       8b 45 0c                mov    0xc(%ebp),%eax
 80483e2:       8b 55 08                mov    0x8(%ebp),%edx
 80483e5:       89 d1                   mov    %edx,%ecx
 80483e7:       29 c1                   sub    %eax,%ecx
 80483e9:       89 c8                   mov    %ecx,%eax
 80483eb:       5d                      pop    %ebp
 80483ec:       c3                      ret

080483ed <main>:
 80483ed:       55                      push   %ebp
 80483ee:       89 e5                   mov    %esp,%ebp
 80483f0:       83 ec 18                sub    $0x18,%esp
 80483f3:       c7 44 24 04 05 00 00    movl   $0x5,0x4(%esp)
 80483fa:       00
 80483fb:       c7 04 24 05 00 00 00    movl   $0x5,(%esp)
 8048402:       e8 d5 ff ff ff          call   80483dc <resta>
 8048407:       89 45 fc                mov    %eax,-0x4(%ebp)
 804840a:       8b 45 fc                mov    -0x4(%ebp),%eax
 804840d:       c9                      leave
 804840e:       c3                      ret
 804840f:       90                      nop
```

Figura 3.23: El código ensamblador del programa de la Figura 3.22

En la Figura 3.24 podemos ver el grafo de control de flujo para el mismo programa, el grafo fue generado mediante la utilidad iltrans. Recordemos que una de las opciones de la utilidad iltrans es generar el grafo de control de flujo de un programa. Además es posible exportar ese grafo a formato DOT y visualizarlo utilizando programas afines. Como podemos notar el grafo esta compuesto por 4 subgrafos. Los nodos BB_Exit y BB_Error son nodos agregados por BAP para indicar el punto en el que termina la ejecución del programa y el punto en el que el programa debería abortar. Notemos que no hay razones que puedan hacer abortar el programa y que BAP no pudo determinar en donde termina la ejecución del programa. Los otros dos subgrafos representan al programa en si. El subgrafo de la derecha corresponde al código del programa luego de la llamada a la función resta como podemos observar en la Figura 3.23, y el grafo de la izquierda corresponde a la función main, junto con el cuerpo de la función resta hasta la instrucción de retorno. Notemos que por defecto BAP realizo el inlining de la función resta, hasta el RET.

A la hora de realizar verificaciones formales sobre el código, recordemos de la sección 2.4.3 que el algoritmo soportado por topredicate para convertir un grafo de control de flujo a GCL solo funciona sobre grafos conexos y sin ciclos. Es por esto que no podremos hacer verificaciones sobre este código hasta que hayamos resuelto el problema de los llamados entre funciones.

En este primer caso en el que a lo sumo se llama una vez a cualquier otra función distinta de main, la solución propuesta es simplemente remplazar la semántica de la instrucción RET. El cambio consiste en remplazar el salto indirecto por la dirección a la que sabemos que el programa retornara (esto solo es posible porque la función es llamada solo una vez). Para el ejemplo de la Figura 3.24, sería remplazar *jmp T_ra_84:u32: @str "ret"* por *jmp 0x8048407.*

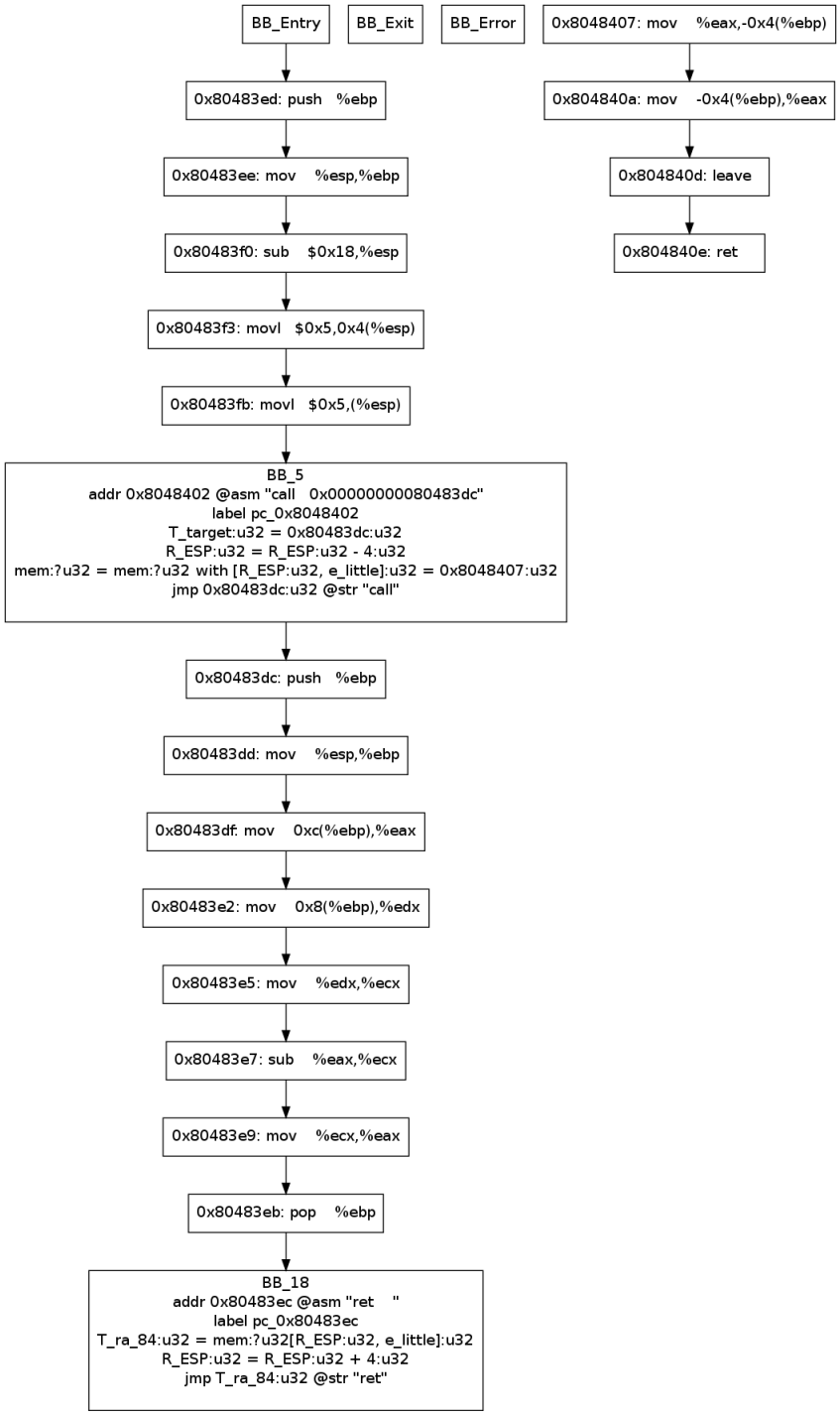


Figura 3.24: Grafo de control de flujo del código ensamblador de la Figura 3.23.

Veamos ahora que sucede cuando existe más de un llamado a la misma función. Notemos que en este caso, ya no es posible determinar la dirección a la que saltará el RET, ya que la función puede ser llamada desde más de un lugar. En la Figura 3.25 podemos ver un ejemplo de un programa que realiza dos llamados a la función suma.

```
#include <stdio.h>
#include <stdlib.h>

int suma(int a, int b){
    return a + b;
}

int main(int argc, char *argv[]) {
    int num;
    int result = suma(num,5);
    int result2 = suma(result,2);
    return result2;
}
```

Figura 3.25: Función con dos llamadas a la misma función.

En la Figura 3.26 podemos observar el código ensamblador de este programa y en la Figura 3.27 podemos observar el grafo de control de flujo.

En el grafo de control de flujo podemos observar un grafo no conexo en el que se ven los dos llamados a la función suma (nodos con direcciones 0x80483fd y 0x8048413). Para convertir al grafo en conexo sería necesario unir el nodo que contiene el RET de la función suma con el nodo con dirección 0x8048402 y con el nodo con dirección 0x8048418, pero entonces nuestro grafo tendría ciclos, sin mencionar que representaría a un programa con distinta semántica que el original (tendría un salto no determinístico). Por lo que no nos serviría para poder realizar las verificaciones.

La solución a este problema fue simplemente copiar la función suma dos veces y remplazar los llamados a las funciones sumas, con llamados a las copias. Al tener las copias, caemos en el caso explicado anteriormente, ya que luego del copiado no existen varios llamados a la misma función, sino que son dos llamados a funciones distintas, y hay exactamente un solo llamado para cada función. Es por esto que podemos proceder a remplazar la semántica del RET para cada copia y unir el grafo correctamente.

```
080483dc <suma>:
 80483dc:       55                      push   %ebp
 80483dd:       89 e5                   mov    %esp,%ebp
 80483df:       8b 45 0c                mov    0xc(%ebp),%eax
 80483e2:       8b 55 08                mov    0x8(%ebp),%edx
 80483e5:       01 d0                   add    %edx,%eax
 80483e7:       5d                      pop    %ebp
 80483e8:       c3                      ret

080483e9 <main>:
 80483e9:       55                      push   %ebp
 80483ea:       89 e5                   mov    %esp,%ebp
 80483ec:       83 ec 18                sub    $0x18,%esp
 80483ef:       c7 44 24 04 05 00 00    movl   $0x5,0x4(%esp)
 80483f6:       00
 80483f7:       8b 45 f4                mov    -0xc(%ebp),%eax
 80483fa:       89 04 24                mov    %eax,(%esp)
 80483fd:       e8 da ff ff ff          call   80483dc <suma>
 8048402:       89 45 f8                mov    %eax,-0x8(%ebp)
 8048405:       c7 44 24 04 02 00 00    movl   $0x2,0x4(%esp)
 804840c:       00
 804840d:       8b 45 f8                mov    -0x8(%ebp),%eax
 8048410:       89 04 24                mov    %eax,(%esp)
 8048413:       e8 c4 ff ff ff          call   80483dc <suma>
 8048418:       89 45 fc                mov    %eax,-0x4(%ebp)
 804841b:       8b 45 fc                mov    -0x4(%ebp),%eax
 804841e:       c9                      leave
 804841f:       c3                      ret
```

Figura 3.26: Código ensamblador del programa de la Figura 3.25.

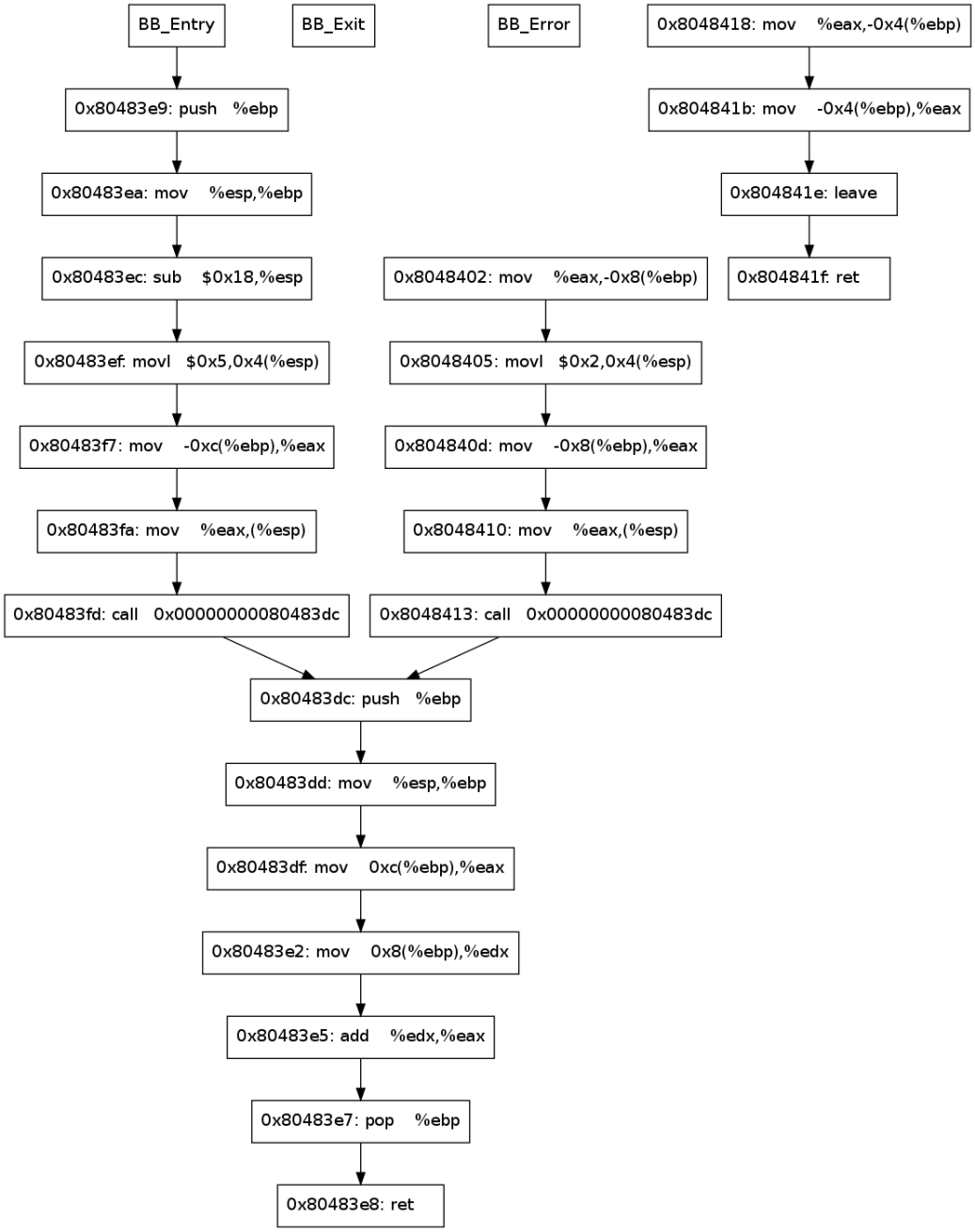


Figura 3.27: Grafo de control de flujo del código de la Figura 3.26.

A continuación veremos como se da la conversión comparando el código intermedio antes y después de realizar el método desarrollado.

En la Figura 3.28 podemos ver el código intermedio del programa antes de ser transformado. Notemos que los dos saltos correspondientes al call de la función suma, líneas 15 y 22 apuntan a la misma dirección.

```
1  addr 0x80483e9 @asm "push   %ebp"
2  label pc_0x80483e9
3  T_t:u32 = R_EBP:u32
4  R_ESP:u32 = R_ESP:u32 - 4:u32
5  mem:?u32 = mem:?u32 with [R_ESP:u32, e_little]:u32 = T_t:u32
6  addr 0x80483ea @asm "mov    %esp,%ebp"
7  label pc_0x80483ea
8  R_EBP:u32 = R_ESP:u32
9  ...
10 addr 0x80483fd @asm "call   0x00000000080483dc"
11 label pc_0x80483fd
12 T_target:u32 = 0x80483dc:u32
13 R_ESP:u32 = R_ESP:u32 - 4:u32
14 mem:?u32 = mem:?u32 with [R_ESP:u32, e_little]:u32 = 0x8048402:u32
15 jmp 0x80483dc:u32 @str "call"
16 ...
17 addr 0x8048413 @asm "call   0x00000000080483dc"
18 label pc_0x8048413
19 T_target_81:u32 = 0x80483dc:u32
20 R_ESP:u32 = R_ESP:u32 - 4:u32
21 mem:?u32 = mem:?u32 with [R_ESP:u32, e_little]:u32 = 0x8048418:u32
22 jmp 0x80483dc:u32 @str "call"
23 ...
24 addr 0x804841f @asm "ret    "
25 label pc_0x804841f
26 T_ra:u32 = mem:?u32[R_ESP:u32, e_little]:u32
27 R_ESP:u32 = R_ESP:u32 + 4:u32
28 jmp T_ra:u32 @str "ret"
29
30 addr 0x80483dc @asm "push   %ebp"
31 label pc_0x80483dc
32 T_t_83:u32 = R_EBP:u32
33 R_ESP:u32 = R_ESP:u32 - 4:u32
34 mem:?u32 = mem:?u32 with [R_ESP:u32, e_little]:u32 = T_t_83:u32
35 addr 0x80483dd @asm "mov    %esp,%ebp"
36 label pc_0x80483dd
37 R_EBP:u32 = R_ESP:u32
38 ...
39 addr 0x80483e7 @asm "pop    %ebp"
40 label pc_0x80483e7
41 R_EBP:u32 = mem:?u32[R_ESP:u32, e_little]:u32
42 R_ESP:u32 = R_ESP:u32 + 4:u32
43 addr 0x80483e8 @asm "ret    "
44 label pc_0x80483e8
45 T_ra_86:u32 = mem:?u32[R_ESP:u32, e_little]:u32
46 R_ESP:u32 = R_ESP:u32 + 4:u32
47 jmp T_ra_86:u32 @str "ret"
```

Figura 3.28: Código intermedio del ensamblador de la Figura 3.26.

Ahora comparemos con la Figura 3.29, que contiene el código intermedio

luego de la transformación.

Entre las líneas 6 y 10, tenemos el primer call a suma, y entre las líneas 14 y 18 el segundo. Notemos que ambos saltos apuntan a labels en vez de a una dirección de memoria como en el código original. El remplazo de direcciones por labels se introdujo para evitar tener que inventar direcciones de memoria inexistentes en el código original que no habrían tenido ninguna relación con el archivo original.

Además, notemos que ambos labels son distintos, el primer call recibió el prefijo func2, mientras que el segundo recibió func3, también notemos que la función main tiene un prefijo.

Para apreciar el inlining de las funciones miremos que el primer call a la función suma (línea 10) realiza un salto al label func2_pc_0x80483dc, función cuyo cuerpo comienza en la línea 28. En el retorno de esta función (línea 37), podemos ver el salto al label func1_pc_0x8048402. Si observamos en la línea 11, ese es el label correspondiente a la instrucción siguiente del primer call a la función suma. Lo mismo pasa con la siguiente llamada a suma.

```
1  label func1_pc_0x80483e9 @asm "push    %ebp"
2  T_t:u32 = R_EBP:u32
3  R_ESP:u32 = R_ESP:u32 - 4:u32
4  mem:?u32 = mem:?u32 with [R_ESP:u32, e_little]:u32 = T_t:u32
5  [...]
6  label func1_pc_0x80483fd @asm "call    0x00000000080483dc"
7  T_target:u32 = 0x80483dc:u32
8  R_ESP:u32 = R_ESP:u32 - 4:u32
9  mem:?u32 = mem:?u32 with [R_ESP:u32, e_little]:u32 = 0x8048402:u32
10 jmp "func2_pc_0x80483dc" @str "call"
11 label func1_pc_0x8048402 @asm "mov     %eax,-0x8(%ebp)"
12 mem:?u32 = mem:?u32 with [R_EBP:u32 + (-(8:u32)), e_little]:u32 = R_EAX:u32
13 [...]
14 label func1_pc_0x8048413 @asm "call    0x00000000080483dc"
15 T_target_81:u32 = 0x80483dc:u32
16 R_ESP:u32 = R_ESP:u32 - 4:u32
17 mem:?u32 = mem:?u32 with [R_ESP:u32, e_little]:u32 = 0x8048418:u32
18 jmp "func3_pc_0x80483dc" @str "call"
19 label func1_pc_0x8048418 @asm "mov     %eax,-0x4(%ebp)"
20 mem:?u32 = mem:?u32 with [R_EBP:u32 + (-(4:u32)), e_little]:u32 = R_EAX:u32
21 [...]
22 label func1_pc_0x804841f @asm "ret     "
23 T_ra:u32 = mem:?u32[R_ESP:u32, e_little]:u32
24 R_ESP:u32 = R_ESP:u32 + 4:u32
25 bof_7:u32 = T_ra:u32
26 jmp "the_end" @str "ret"
27
28 label func2_pc_0x80483dc @asm "push    %ebp"
29 T_t_83:u32 = R_EBP:u32
30 R_ESP:u32 = R_ESP:u32 - 4:u32
31 mem:?u32 = mem:?u32 with [R_ESP:u32, e_little]:u32 = T_t_83:u32
32 [...]
33 label func2_pc_0x80483e8 @asm "ret     "
34 T_ra_86:u32 = mem:?u32[R_ESP:u32, e_little]:u32
35 R_ESP:u32 = R_ESP:u32 + 4:u32
36 bof_6:u32 = T_ra_86:u32
37 jmp "func1_pc_0x8048402" @str "ret"
38
39 label func3_pc_0x80483dc @asm "push    %ebp"
40 T_t_83:u32 = R_EBP:u32
41 R_ESP:u32 = R_ESP:u32 - 4:u32
42 mem:?u32 = mem:?u32 with [R_ESP:u32, e_little]:u32 = T_t_83:u32
43 [...]
44 label func3_pc_0x80483e8 @asm "ret     "
45 T_ra_86:u32 = mem:?u32[R_ESP:u32, e_little]:u32
46 R_ESP:u32 = R_ESP:u32 + 4:u32
47 bof_5:u32 = T_ra_86:u32
48 jmp "func1_pc_0x8048418" @str "ret"
49
50 label the_end
51 halt true
```

Figura 3.29: Código intermedio luego de ser procesado.

Ahora que ya explicamos como es el método para solucionar el problema del análisis interprocedural, veamos el código desarrollado.

Para el copiado de las funciones se definió la función copy_function, podemos ver su signatura en la Figura 3.30.

```
def copy_function(data, functions, ret_label, relocations, func_name)
```

Figura 3.30: Signatura de la función copy_function.

La función toma como argumentos:

- data: el código intermedio de la función a copiar.
- functions: un diccionario con todas las funciones que son llamadas de la función a copiar.
- ret_label: el label con el que remplazaremos la dirección de retorno.
- relocations: un diccionario con todas las funciones importadas del binario analizado.
- func_name: el label que será utilizado para distinguir con unicidad a la copia de la función.

La función recorre cada una de las instrucciones del código intermedio, y dependiendo del tipo de instrucción realiza las modificaciones acordes para que la copia de la función sea correcta.

Los tipos de instrucciones para los que se hace necesario modificar el código intermedio son: los labels, saltos condicionales y saltos incondicionales. Veamos los cambios necesarios para cada uno de ellos.

Si la instrucción es un label de tipo StrLabel, simplemente lo remplazamos por otro que tiene como prefijo el nombre la función, como podemos observar en la Figura 3.31. Notemos que estamos omitiendo los labels de tipo AddrLabel, esto lo hacemos porque al crear un AddrLabel es necesario hacerlo con una dirección de memoria. En el código intermedio original, la dirección de memoria indica en donde se cargaron los bytes de esa instrucción en memoria, pero al crear una copia de la función, como no es posible usar la misma dirección nuevamente, deberíamos utilizar una dirección de memoria que no tendría ninguna relación con el programa en si. Es por esto que se opto por eliminar estos tipos de labels.

Si la instrucción es de tipo salto condicional, recordemos que el destino de los saltos es un AddrLabel o un StrLabel. Como estamos cambiando todos los

```
1  elif 'label' in str(stmt):
2      new_label = StrLabel('{0}_{1}'.format(func_name ,
3                                            stmt.label.name))
4      stmt.label = new_label
5      stmt.attrs.append(asm)
6      instrs.append(stmt)
```

Figura 3.31: Caso label en el código de copy_functions.

nombres de los StrLabel y eliminando los AddrLabel es necesario modificar los saltos condicionales acordemente.

Como podemos observar en la Figura 3.32, modificamos los dos posibles destinos del salto condicional. Y, en cada caso, chequeamos el tipo del label para proceder. Si el label es un StrLabel entonces procedemos como en el caso que vimos anteriormente, mientras que si es un AddrLabel lo "convertimos" a un StrLabel agregándole el prefijo del nombre de la función y la dirección de memoria correspondiente.

```
1  elif str(stmt).startswith('cjmp'):
2      if type(stmt.iftrue) == type(Lab()):
3          stmt.iftrue.string = '{0}_{1}'.format(func_name,
4                                                stmt.iftrue.string)
5      else:
6          address = stmt.iftrue.inte
7          new_lab = Lab('{0}_pc_0x{1:x}'.format(func_name, address))
8          stmt.iftrue = new_lab
9
10     if type(stmt.iffalse) == type(Lab()):
11         stmt.iffalse.string = '{0}_{1}'.format(func_name,
12                                                stmt.iffalse.string)
13     else:
14         address = stmt.iffalse.inte
15         new_lab = Lab('{0}_pc_0x{1:x}'.format(func_name, address))
16         stmt.iffalse = new_lab
17
18     instrs.append(stmt)
```

Figura 3.32: Caso salto condicional en el código de copy_functions.

Finalmente, si la instrucción es de tipo salto incondicional, tenemos dos grandes casos, saltos directos y saltos indirectos. A su vez, dentro de los saltos indirectos tenemos el salto ocasionado por la instrucción RET y los saltos que surgen de las funciones importadas.

En esta sección solamente nos interesan dos casos: el de los saltos directos

y el del salto indirecto ocasionado por la instrucción RET. Para este último caso, si encontramos un salto indirecto producido por RET, simplemente remplazamos la expresión del salto original por el label ret_label que toma la función copy_functions como parámetro. Esto podemos verlo en la línea 20 de la Figura 3.33.

En el otro caso, el salto se da hacia una dirección de memoria fija, por lo que tomamos la dirección, nos fijamos en el diccionario de funciones para obtener el código intermedio de esa función, y llamamos recursivamente a la función copy_function sobre el código de la función. Luego remplazamos el destino del salto con un label que apunta a la copia de la función (línea 13) y finalmente agregamos la función a una lista de funciones copiadas. El código intermedio de todas las funciones copiadas es agregado luego al código intermedio de la función original modificada.

```
1  elif type(stmt) == type(Jmp()):
2      # If there is a direct jump to a function we inline it
3      if type(stmt.exp) == type(Int(0)):
4          address = stmt.exp.inte
5          if address in functions:
6              new_func_name = 'call_%s' %random_name
7              label, new_function = copy_function(deepcopy(functions[address]),
8                                                  functions,
9                                                  get_ret_label(last_label),
10                                                 relocations,
11                                                 new_func_name)
12
13             stmt.exp = Lab(label)
14             instrs.append(stmt)
15
16             copied_functions.append(new_function)
17     else:
18         if 'ret' in str(stmt):
19             [...]
20             stmt.exp = Lab(ret_label)
21             instrs.append(stmt)
```

Figura 3.33: Caso salto incondicional en el código de copy_functions.

Notemos de la línea 9 de la Figura 3.33, que el label de retorno de la copia de la función es calculado mediante la función get_ret_label utilizando como label a last_label. Esta variable, almacena el label de la instrucción ensamblador que esta siendo procesada en el momento.

Para entender porque es necesaria esta variable observemos el lenguaje intermedio de la instrucción CALL en la Figura 3.34. Veamos que el label

de esta instrucción ensamblador esta en la primera instrucción, pero el salto esta en la ultima instrucción del código intermedio. Recordemos que una instrucción ensamblador esta compuesta por varias instrucciones del código intermedio. El código desarrollado va procesando instrucción por instrucción, por lo que cuando lleguemos a procesar el salto ya no tendremos ninguna referencia del label del comienzo. Para eso creamos la variable last_label que guarda en todo momento, el label de la instrucción ensamblador en cuestión.

```
label call4_pc_0x8048413 @asm "call    0x00000000080483dc"
T_target_81:u32 = 0x80483dc:u32
R_ESP:u32 = R_ESP:u32 - 4:u32
mem:?u32 = mem:?u32 with [R_ESP:u32, e_little]:u32 = 0x8048418:u32
jmp "call3_pc_0x80483dc" @str "call"
```

Figura 3.34: Call completo en el lenguaje intermedio

Ahora veamos porque necesitamos esa referencia al label, para ello observemos el código de get_ret_label en la Figura 3.35. Este código toma el label que le pasamos como parámetro e incrementa la dirección a la que apunta el label en 5. El incremento en 5 se debe a que una instrucción call en ensamblador ocupa 5 bytes de memoria, por lo que la dirección de memoria de la próxima instrucción luego de un call siempre será la dirección del call incrementada en 5.

```
def get_ret_label(label):
    name,_,address = label.split('_')
    return '%s_pc_0x%x' %(name, int(address,16) + 5)
```

Figura 3.35: Código de get_ret_label

Por lo tanto, cuando encontramos el salto producido por un call, necesitamos saber el label de la instrucción ensamblador para poder así calcular el label de la próxima instrucción al call, que será la dirección de retorno de la función llamada.

Una cosa a tener en cuenta es que el método desarrollado no soporta funciones recursivas o funciones con recursividad mutua. Para evitar analizar funciones de este tipo se mantiene una lista que almacena las direcciones de las funciones que fueron llamadas anteriormente. Si la función a la que llamamos, se encuentra en esta lista el programa aborta. El código desarrollo para detectar funciones recursivas puede verse en la Figura 3.36.

```
func_address = data[0].label.addr

# Recursive call detection
if func_address in function_call_stack:
    print "[*] Recursive call detected, \
           we don't support recursive calls"
    print '[*] Printing call stack'
    function_call_stack.append(func_address)
    for depth,elem in enumerate(function_call_stack):
        print '[*] %s 0x%x' %('\t' * depth, elem)
    sys.exit(1)
else:
    function_call_stack.append(func_address)
```

Figura 3.36: Código para detectar funciones recursivas.

3.4. Aproximación a las funciones importadas

Al convertir el código de un binario al lenguaje intermedio nos topamos con el problema de las funciones importadas. Estas, son funciones utilizadas por el código pero cuya implementación no es distribuida junto con el código, sino que reside en bibliotecas comunes del sistema. Cuando corremos el programa, las dependencias de funciones importadas se resuelven dinámicamente antes de comenzar la ejecución del programa.

Veamos ahora como funciona la resolución de las funciones importadas. Cuando un binario se compila utilizando funciones de una biblioteca dinámica, el compilador agrega una tabla de relocaciones en los headers del binario. Una tabla de relocaciones es básicamente una tabla que contiene punteros a las funciones importadas. Al compilarse el programa, como el compilador no puede saber en que dirección de memoria se encontraran cargadas las funciones importadas que el programa utiliza, lo que hace es remplazar el llamado a una función importada por un salto indirecto o llamado indirecto dentro de la tabla de relocaciones. Cuando el programa es cargado en memoria, la tabla de relocaciones es llenada con las direcciones de las funciones reales y recién entonces puede comenzar la ejecución del programa. Ahora veremos dos ejemplos de como se lleva a cabo este proceso para programas compilados con gcc y Visual Studio.

La forma de manejar las funciones importadas por parte de un programa depende del compilador que se haya utilizado para crear ese programa. Se eligieron estos ya que ambos son compiladores muy utilizados en las plataformas respectivas, gcc en Linux y Visual Studio en Windows.

En la Figura 3.37, podemos ver como sería un llamado a una función importada en un programa compilado con gcc. En la línea 3 podemos observar una llamada a la función open. más abajo, en la línea 7, el salto indirecto hacia el contenido de la dirección 0x804a00. Esta dirección contendrá la dirección real de la función open que será resuelta dinámicamente al cargarse el programa en memoria.

Notemos como en este caso el compilador crea una función auxiliar que termina llamando a la función real, para cada una de las funciones importadas.

```
1  0804848c <main>:
2   [...]
3   80484bf:    e8 bc fe ff ff      call 8048380 <open@plt>
4   [...]
5
6  08048380 <read@plt>:
7   8048380:    ff 25 0c a0 04 08   jmp  DWORD PTR ds:0x804a00c
```

Figura 3.37: Llamado a función importada en Linux.

En la Figura 3.38 podemos observar como son los llamados a funciones importadas en programas compilados con Visual Studio. A diferencia de los programas compilados con gcc, que llaman a una nueva función que realiza un salto indirecto hacia la función importada, los programas compilados con Visual Studio directamente realizar un call indirecto, como podemos observar en la línea 3.

```
1  0401040 <main>:
2   [...]
3   0402b67:      ff 15 0c a0 40 00     call   DWORD PTR ds:0x40a00c
4   [...]
```

Figura 3.38: Llamado a función importada en Windows.

Para obtener un listado de las funciones importadas se utiliza la función get_relocations que podemos en ver en la Figura 3.39. Esta función toma como entrada el path del binario, recorre las tablas de importaciones del mismo y nos devuelve un diccionario con los nombres de las funciones importadas y la dirección de memoria en donde estará almacenada la dirección real de la función importada. Notemos que la función soporta tanto binarios de Linux como de Windows.

Como dijimos anteriormente el código binario de las funciones importadas no se distribuye junto con el código binario del programa en sí. Es por esto que BAP no puede realizar la conversión al lenguaje intermedio de las mismas

```
def get_relocations(filename):
    """
    Return a dict with the relocations contained in a file
    Taken and modified from
    https://github.com/eliben/pyelftools/blob/master/scripts/readelf.py
    """
    print '[*] Getting relocations'
    relocations = {}
    if not filename:
        print '[*] Relocations not found'
        return relocations

    try:
        pe = pefile.PE(filename)
        for entry in pe.DIRECTORY_ENTRY_IMPORT:
            for imp in entry.imports:
                relocations[imp.address] = imp.name

    except pefile.PEFormatError:
        with file(filename,'r') as fd:
            elffile = ELFFile(fd)

            has_relocation_sections = False
            for section in elffile.iter_sections():
                if not isinstance(section, RelocationSection):
                    continue

                has_relocation_sections = True
                # The symbol table section pointed to in sh_link
                symtable = elffile.get_section(section['sh_link'])

                for rel in section.iter_relocations():
                    offset = rel['r_offset']

                    symbol = symtable.get_symbol(rel['r_info_sym'])
                    # Some symbols have zero 'st_name', so instead
                    # what's used is the name of the section they point
                    # at
                    if symbol['st_name'] == 0:
                        symsec = elffile.get_section(symbol['st_shndx'])
                        symbol_name = symsec.name
                    else:
                        symbol_name = symbol.name
                        relocations[offset] = bytes2str(symbol_name)
    print '[*] Getting relocations DONE'
    return relocations
```

Figura 3.39: Código para obtener las funciones importadas.

y por lo tanto cualquier análisis sobre un programa de este tipo no podrá llevarse a cabo. Cabe aclarar que la mayoría de los programas funcionan de esta forma.

Para resolver este problema, es necesario remplazar todas las funciones importadas de un programa por su equivalente semántico en el código intermedio de BAP.

Para llevar esto acabo, se procedió de la siguiente manera.

1. Programación de código binario con una semántica similar a la de la función importada.

2. Conversión de este código binario al lenguaje intermedio de BAP.

3. Remplazo de las llamadas a la función importada por el código con semántica equivalente obtenido en 2.

Ya que realizar este procedimiento para cada una de las funciones importadas es una tarea que conlleva mucho tiempo, solo se la realizó para las funciones importadas más relevantes para nuestro caso. Las funciones importadas que fueron desarrolladas son las siguientes:

- read
- recv
- memcpy
- strcpy

Una forma directa de resolver el problema de las funciones importadas es tomar la función externa, obtener el código intermedio de la función y utilizarlo. Hacer esto tiene el problema de que muchas de estas funciones terminan llamando al kernel de alguna forma, o llaman a otras funciones que pueden llegar a ser muy complejas y no son estrictamente necesarias a la hora de probar la presencia de buffer overflow. Es por esto que se decidió escribir manualmente un equivalente semántico de cada función. Cabe aclarar que las funciones implementadas tendrán solo la funcionalidad de la función original que sea relevante para detectar buffer overflow, por lo que no tendrán una semántica idéntica a la de la función original.

En la Figura 3.41 podemos ver el código ensamblador de la función main de un programa que realiza un llamado a read. Podemos observar el llamado a read en la línea 24. En la Figura 3.40 podemos ver la definición de la función read.

Notemos que read toma 3 argumentos, esos argumentos son:

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

Figura 3.40: Definición de la función read.

- fildes: descriptor de archivo de donde se obtendrán los datos a copiar.
- buf: dirección de destino
- nbyte: cantidad de bytes a copiar

Recordemos, de la sección 2.2.2.2, que los argumentos de una función son pasados utilizando el stack. En el caso de read los argumentos se almacenan en las direcciones ESP, ESP + 4 y ESP + 8 respectivamente.

Es necesario tener en cuenta las direcciones de memoria en la que serán pasados los argumentos a la hora de implementar el equivalente semántico de la función importada. Notar que los argumentos se guardan en memoria consecutivamente y en orden. Así, el primer argumento (contando desde la izquierda de la definición), es almacenado en ESP, el segundo en ESP + 4, el tercero en ESP + 8, y así sucesivamente.

Podemos ver el almacenamiento del primer argumento en la línea 23 de la Figura 3.41, el segundo en la línea 21 y el tercero en la línea 18.

Como dijimos anteriormente, uno de los argumentos de la función read es un descriptor de archivo. El uso de descriptores de archivo es una característica netamente dinámica de los programas. Como los descriptores de archivo son asignados por el sistema operativo, no es posible determinar que descriptor de archivo será devuelto por las funciones open, connect, etc. Es por esto que debemos encontrar una forma de remplazar el uso de descriptores de archivo que se adapte al método desarrollado.

En remplazo de los descriptores de archivo, utilizaremos un espacio de memoria que sabemos que el programa no utilizará por encontrarse dentro del espacio de memoria asignado al kernel. Trataremos a los archivos, sockets u otras entradas de datos como pedazos de memoria relativos a una dirección base. Es decir, utilizaremos el espacio de memoria como si fuera un buffer en el que los elementos tienen un tamaño fijo. Así, a la primera llamada a alguna función de entrada de datos le será asignada la dirección 0x80000000, a la segunda la dirección 0x80000000 + offset fijo, y así sucesivamente. Es posible modificar ambas opciones directamente desde el código.

En este espacio de memoria no habrá datos definidos, por lo que esa memoria será considerada como simbólica a la hora de razonar con el solver. El valor que el solver le asigne a esta memoria dependerá de la postcondición ingresada y del uso que haga el programa de esa memoria.

```
1  0804848c <main>:
2   804848c:   55                          push   ebp
3   804848d:   89 e5                       mov    ebp,esp
4   804848f:   83 e4 f0                    and    esp,0xfffffff0
5   8048492:   83 ec 30                    sub    esp,0x30
6   8048495:   8b 45 0c                    mov    eax,DWORD PTR [ebp+0xc]
7   8048498:   89 44 24 1c                 mov    DWORD PTR [esp+0x1c],eax
8   804849c:   65 a1 14 00 00 00           mov    eax,gs:0x14
9   80484a2:   89 44 24 2c                 mov    DWORD PTR [esp+0x2c],eax
10  80484a6:   31 c0                       xor    eax,eax
11  80484a8:   c7 44 24 20 00 00 00        mov    DWORD PTR [esp+0x20],0x0
12  80484af:   00
13  80484b0:   c7 44 24 04 00 00 00        mov    DWORD PTR [esp+0x4],0x0
14  80484b7:   00
15  80484b8:   c7 04 24 a8 85 04 08        mov    DWORD PTR [esp],0x80485a8
16  80484bf:   e8 bc fe ff ff              call   8048380 <open@plt>
17  80484c4:   89 44 24 20                 mov    DWORD PTR [esp+0x20],eax
18  80484c8:   c7 44 24 08 1c 00 00        mov    DWORD PTR [esp+0x8],0x1c
19  80484cf:   00
20  80484d0:   8d 44 24 28                 lea    eax,[esp+0x28]
21  80484d4:   89 44 24 04                 mov    DWORD PTR [esp+0x4],eax
22  80484d8:   8b 44 24 20                 mov    eax,DWORD PTR [esp+0x20]
23  80484dc:   89 04 24                    mov    DWORD PTR [esp],eax
24  80484df:   e8 6c fe ff ff              call   8048350 <read@plt>
25  80484e4:   89 44 24 24                 mov    DWORD PTR [esp+0x24],eax
26  80484e8:   b8 00 00 00 00              mov    eax,0x0
27  80484ed:   8b 54 24 2c                 mov    edx,DWORD PTR [esp+0x2c]
28  80484f1:   65 33 15 14 00 00 00        xor    edx,DWORD PTR gs:0x14
29  80484f8:   74 05                       je     80484ff <main+0x73>
30  80484fa:   e8 61 fe ff ff              call   8048360 <__stack_chk_fail@plt>
31  80484ff:   c9                          leave
32  8048500:   c3                          ret
```

Figura 3.41: Código ensamblador de una función que llama a read.

En la figura 3.42 podemos ver el código ensamblador equivalente para la función read. Para cada una de las funciones implementadas se procedió de la siguiente manera:

1. El código ensamblador es compilado utilizando el ensamblador nasm.

2. Se obtiene el código intermedio del binario utilizando la utilidad toil.

3. Se realizan las modificaciones pertinentes sobre el código intermedio. En la Figura 3.43 podemos observar el código intermedio ya modificado para la función read. Más adelante se explicarán cada una de las modificaciones.

4. Se utiliza nuevamente la utilidad toil para exportar a formato JSON el código intermedio modificado. El archivo debe tener el mismo nombre que la función a la que esta implementando.

5. El archivo JSON obtenido es colocado en el directorio asm que se encuentra en el directorio raíz del código desarrollado.

```
    SECTION .text
    global main
main:
    push eax            ; save state
    push ecx
    push edi
    push esi
copyLoop:
    mov edi, [esp + 0x18]       ; destination address
    mov ecx, [esp + 0x1c]       ; byte count
    mov al,[esi]
    mov [edi],al
    inc esi
    inc edi
    dec ecx
    cmp ecx,0
    jnz copyLoop
    pop eax         ; restore state
    pop ecx
    pop edi
    pop esi
    ret
```

Figura 3.42: Código ensamblador para la función read.

Notemos que durante el desarrollo del código de la función read, no se tuvo en cuenta el valor del descriptor de archivo pasado (argumento que se encuentra en la dirección [ESP + 0x20]). El valor de memoria de donde serán copiados los datos es agregado directamente al código intermedio de la función open, como podemos ver en la línea 10 de la Figura 3.43. La constante SOURCE_ADDRESS será remplazada por una dirección de memoria dentro del buffer con el que remplazamos a los descriptores de archivos más tarde por el código desarrollado.

```
1  addr 0x1c @asm "push    %eax"
2  label pc_0x1c
3  T_t_83:u32 = R_EAX:u32
4  R_ESP:u32 = R_ESP:u32 - 4:u32
5  mem:?u32 = mem:?u32 with [R_ESP:u32, e_little]:u32 = T_t_83:u32
6  [...]
7  addr 0x20 @asm "mov     0x18(%esp),%edi"
8  label pc_0x20
9  R_EDI:u32 = mem:?u32[R_ESP:u32 + 0x18:u32, e_little]:u32
10 source_address:u32 = SOURCE_ADDRESS:u32
11 R_ESI:u32 = source_address:u32
12 addr 0x24 @asm "mov     0x1c(%esp),%ecx"
13 label pc_0x24
14 R_ECX:u32 = mem:?u32[R_ESP:u32 + 0x1c:u32, e_little]:u32
15 [...]
16 addr 0x38 @asm "ret     "
17 label pc_0x38
18 T_ra:u32 = mem:?u32[R_ESP:u32, e_little]:u32
19 R_ESP:u32 = R_ESP:u32 + 4:u32
20 jmp T_ra:u32 @str "ret"
```

Figura 3.43: Código intermedio de la función read ya modificado.

Ahora analizaremos el código desarrollado para solucionar el problema de las funciones importadas. La función encargada de realizar la copia de las funciones importadas es copy_imported_function, está toma como argumentos:

- address: la dirección de la función llamada.
- functions: un diccionario con todas las funciones que son llamadas de la función a copiar.
- ret_label: el label con el que remplazaremos la dirección de retorno.

- relocations: un diccionario con todas las funciones importadas del binario analizado.
- random_name: el label que será utilizado para distinguir con unicidad a la copia de la función.

La función toma la dirección de la función importada y busca el nombre de la función en el diccionario de relocaciones, como podemos observar en la línea 5 de la Figura 3.44. En caso de estar implementada esa función, se procede de la siguiente manera:

1. Se lee el contenido del archivo.
2. Se remplaza la dirección de SOURCE_ADDRESS por el valor de la variable global input_address (líneas 12 y 13). Como se explicó anteriormente este valor es utilizado para la obtención de datos simbólicos.
3. Se parsea la función leída, ya que recordemos que está en formato JSON.
4. Finalmente se procede a realizar la copia de la misma y se imprime en pantalla el nombre de la función y la dirección de memoria de donde se tomarán los datos para facilitar luego la interpretación de los resultados del solver.

Es importante tener en mente que los dos cuestiones más importantes que tenemos que resolver al tratar con las funciones importadas es:

1. Detectar cuando hay una función importada.
2. Obtener la dirección de la tabla de relocaciones en la que se encuentra el puntero a la función real.

Necesitamos saber cuando estamos ante una función importada en el código binario para saber cuando realizar los análisis pertinentes. Sobre el segundo punto, recordemos que el diccionario de importaciones que obtenemos con la función get_relocations contiene direcciones como claves, y los nombres de las funciones importadas como valores. Necesitamos saber la dirección dentro de la tabla de relocaciones para determinar a que función esta llamando el binario.

Ahora veremos el código encargado de detectar los llamados a funciones importadas. Pero primero veamos como es la representación en código intermedio para cada uno de los llamados a funciones importadas que vimos anteriormente.

```
1  def copy_imported_function(address, functions, ret_label,
2                             relocations, random_name):
3      global input_address
4
5      function_path = './asm/%s.json' %relocations[address]
6
7      try:
8          function_data = open(function_path).read()
9      except:
10         return
11
12     source_address = input_address + BUFF_SIZE
13     function_data.replace('SOURCE_ADDRESS', str(source_address))
14     function_data = json.loads(function_data)
15
16     input_address += BUFF_SIZE
17     imported_function = [parse_statement(stmt) for stmt in function_data]
18     label, new_function = copy_function(imported_function,
19                                         functions,
20                                         ret_label,
21                                         relocations,
22                                         '%s_%s' %(relocations[address],
23                                                   random_name))
24
25     print '[*] %s is taking data from 0x%x' %(label, source_address)
26     return label, new_function
```

Figura 3.44: Función de copiado de funciones externas.

En la Figura 3.45 podemos observar como sería la representación en el lenguaje intermedio del llamado de la Figura 3.37. Recordemos que este programa fue compilado con gcc.

```
1  [...]
2  addr 0x80484bf @asm "call    0x0000000008048380"
3  label pc_0x80484bf
4  T_target_82:u32 = 0x8048380:u32
5  R_ESP:u32 = R_ESP:u32 - 4:u32
6  mem:?u32 = mem:?u32 with [R_ESP:u32, e_little]:u32 = 0x80484c3:u32
7  jmp 0x8048380:u32 @str "call"
8  [...]
9  addr 0x8048380 @asm "jmp     *0x804a00c"
10 label pc_0x8048380
11 jmp mem:?u32[0x804a00c:u32, e_little]:u32
12 [...]
```

Figura 3.45: Código intermedio de una llamada a función importada en Linux.

Para detectar un llamado a una función importada para este caso, buscamos el string 'jmp mem?:u32' en la representación de la instrucción, como podemos observar en la línea 2 de la Figura 3.47. Buscamos ese string porque esa es la representación de un salto indirecto en el lenguaje intermedio.

En la Figura 3.46 podemos ver el lenguaje intermedio para el código de la Figura 3.38.

```
1  addr 0x402b67 @asm "call    *0x40a00c"
2  label pc_0x402b67
3  T_target_104:u32 = mem:?u32[0x40a00c:u32, e_little]:u32
4  R_ESP:u32 = R_ESP:u32 - 4:u32
5  mem:?u32 = mem:?u32 with [R_ESP:u32, e_little]:u32 = 0x402b6d:u32
6  jmp T_target_104:u32 @str "call"
```

Figura 3.46: Código intermedio de una llamada a función importada en Windows.

Para este otro caso, el de programas compilados con Visual Studio, lo que hacemos es buscar el string 'T_target' en la representación de la instrucción, como podemos observar en la línea 30 de la Figura 3.47. Cuando encontremos este string, solucionamos el primer problema que planteamos anteriormente pero aun tenemos el problema de determinar la dirección de memoria en donde se encuentra la función real, ya que la dirección no aparece en la instrucción del salto como podemos observar en la Figura 3.46. Si observamos la línea 3 de la misma figura, podemos ver que la dirección de memoria que nos interesa se encuentra allí. Para obtener la direccion que necesitamos, utilizamos una variable T_target que almacena las instrucciones de lenguaje intermedio de tipo asignación que tienen como destino cualquier variable con nombre T_target. Como el salto indirecto se encuentra luego de la asignación a la variable T_target, al momento de procesar la instrucción del salto, la variable T_target ya tendrá la dirección deseada. Luego utilizamos esa variable para obtener la dirección de memoria que necesitamos.

Notemos que en ambos casos, luego de encontrar una función importada, lo primero que hacemos es guardar la dirección de la función y luego llamar a copy_imported_function. Si la función importada es una de las funciones que implementamos entonces simplemente realizamos la copia y la agregamos en el lugar como podemos observar en las líneas 12 a 15 para el caso de funciones compiladas con gcc y líneas 41 a 47 para el otro caso de la Figura 3.47.

Caso contrario, tenemos que arreglar el estado del stack. Al realizarse el CALL, el valor de ESP fue decrementado en 4, y, al no existir función de retorno para la función importada, es necesario incrementar en 4 manualmente el valor de ESP, y además volver el flujo de ejecución a la función llamadora. Esto podemos verlo en las líneas 17 a 27 y 49 a 59.

Notemos también en las líneas 34 y 35, el uso de un contador global para asegurar la unicidad de los nombres de las copias de las funciones.

```
# Linux relocations
elif 'jmp mem:?u32' in str(stmt):
    address = stmt.exp.address.inte
    if address in relocations:
        counter = counter + 1
        new_func_name = 'fun%s' %str(counter)
        copied_function = copy_imported_function(address,
                                                 functions,
                                                 ret_label,
                                                 relocations,
                                                 new_func_name)
        if copied_function:
            label, new_function = copied_function
            for instr in new_function:
                instrs.append(instr)
        else:
            # If its not defined fix ESP value.
            # ESP = ESP + 4
            var = Variable(1, 'R_ESP', Register(32))
            exp = BinOp('plus',
                        Variable(1, 'R_ESP', Register(32)),
                        Int(4,Register(32))
                        )
            instrs.append(Move(var, exp))
            # And replace the jmp address with the corresponding label
            stmt.exp = Lab(ret_label)
            instrs.append(stmt)

# Windows relocations
elif 'T_target' in str(stmt):
    if type(T_target.exp) == type(Int(0)):
        address = T_target.exp.address.inte
        if address in relocations:
            counter = counter + 1
            new_func_name = 'fun%s' %str(counter)
            copied_function = copy_imported_function(address,
                                                     functions,
                                                     ret_label,
                                                     relocations,
                                                     new_func_name)
            if copied_function:
                label, new_function = copied_function
                # Remove the last jmp from the ret
                # We are inlining the function so we don't need to
                # jump anywhere.
                for instr in new_function[:-1]:
                    instrs.append(instr)
            else:
                # If its not defined fix ESP value.
                # ESP = ESP + 4
                var = Variable(1, 'R_ESP', Register(32))
                exp = BinOp('plus',
                            Variable(1, 'R_ESP', Register(32)),
                            Int(4,Register(32))
                            )
                instrs.append(Move(var, exp))
                # And replace the jmp address with the corresponding labe
                stmt.exp = Lab(ret_label)
                instrs.append(stmt)
# If none of the above cases happens just append the
# instruction without any changes.
else:
    instrs.append(stmt)
```

Figura 3.47: Inlining de funciones externas.

3.5. Chequeo de buffer overflow

Para realizar el chequeo de buffer overflow utilizaremos la utilidad `topredicate`. Recordemos que esta utilidad tiene la capacidad de convertir un programa escrito en el lenguaje intermedio de BAP a GCL, para luego poder calcular la precondición más débil del programa y realizar verificaciones sobre el mismo. Las verificaciones se realizan utilizando la fórmula de la precondición más débil junto con SMT solvers.

Dado un programa S, y una postcondición Q, sabemos que $wp.S.Q$ es la precondición más débil que debe cumplirse al comienzo de la ejecución de S, para que cuando la ejecución termine se cumpla Q.

Como sabemos, la gran mayoría de los programas funcionan a partir de datos de entrada, estos pueden ser obtenidos de un archivo, a través de un socket, etc. Si un programa S tiene como datos de entrada $X_1, X_2, \ldots, X_n$, entonces estas variables serán variables libres dentro de $wp.S.Q$. Por lo tanto, que pueda cumplirse Q luego de la ejecución del programa dependerá de los valores que puedan adoptar estas variables.

Recordemos que la postcondición nos debe permitir decir si existe buffer overflow en el programa S, sobrescribiendo alguna dirección de retorno con datos controlados por el usuario.

En nuestro caso, los datos controlados por el usuario estarán representados por las variables libres. Para ser más específicos, será la memoria obtenida de las llamadas a funciones de entrada de datos como recv, read, get, etc. La posibilidad de sobrescribir una dirección de retorno dependerá del valor de estos datos y de si esos datos son copiados en el stack.

Ya definimos la existencia de buffer overflow como la posibilidad de sobrescribir cualquiera de las direcciones de retorno de un programa. Definiremos la postcondición de la siguiente manera:

- Por cada dirección de retorno que tenga el programa crearemos una variable bof_i que almacenará el valor de la dirección de retorno. Explicaremos más detalladamente esto más adelante.

- Elegiremos un valor arbitrario con el que desearíamos sobrescribir alguna de las direcciones de retorno, al que llamaremos val. Hay que tener en cuenta que este valor debe ser elegido de modo de que no coincida con alguna de las direcciones de retorno propias del programa. Podemos tomar, por ejemplo, una dirección que este fuera del espacio de memoria donde se mapea el programa.

A continuación explicaremos como se realiza la definición de las variables bof_i.

En la Figura 3.48 podemos observar el código intermedio de una instrucción de tipo RET. Notemos en la línea 3, que la dirección de retorno de la función es almacenada en la variable T_ra.

```
1 addr 0x4011de @asm "ret      "
2 label pc_0x4011de
3 T_ra:u32 = mem:?u32[R_ESP:u32, e_little]:u32
4 R_ESP:u32 = R_ESP:u32 + 4:u32
5 jmp T_ra:u32 @str "ret"
```

Figura 3.48: Instrucción RET original.

Para almacenar este valor se crea una variable auxiliar bof_i, que guardara el valor de la variable T_ra. En la Figura 3.49 podemos observar en la línea 4 un ejemplo de una instrucción ret modificada. Notemos la asignación de la variable T_ra a la nueva variable bof_4.

```
1 label func5_pc_0x4011de @asm "ret      "
2 T_ra:u32 = mem:?u32[R_ESP:u32, e_little]:u32
3 R_ESP:u32 = R_ESP:u32 + 4:u32
4 bof_4:u32 = T_ra:u32
5 jmp "the_end" @str "ret"
```

Figura 3.49: Instrucción RETt modificada.

Suponiendo que el programa tiene n direcciones de retorno, la postcondición resultante será:

$$bof_1 = val \vee bof_2 = val \vee \ldots \vee bof_n = val$$

Ahora analicemos porque esta postcondición nos permite detectar si existe un buffer overflow en el programa.

En un programa que no es vulnerable a buffer overflow, esta postcondición siempre será falsa ya que no hay ninguna manera de que una dirección de retorno pueda tomar el valor de val.

Ahora, si el programa es vulnerable a buffer overflow, es posible que podamos sobrescribir alguna de las direcciones de retorno, por lo cual es posible que $bof_i = val$ para algún valor de i. Esto dependerá en ultima instancia de los datos de entrada que tome el programa, es decir, de nuestras variables libres.

Un posible inconveniente con esta postcondición surge del hecho que la función a partir de la cual comenzamos el análisis no es llamada por ninguna otra función. Si bien en la ejecución normal del programa esta función es llamada por otra función (en el stack podremos encontrar la dirección de memoria a la que debe retornar la función una vez terminada), en lo que respecta a nuestra tarea de análisis, esta función no es llamada por nadie. Esto quiere decir que al ejecutarse la instrucción RET de la función, el programa tomará un valor de una dirección del stack que en realidad no está definida, lo cual implica que la dirección de retorno de cualquier función inicial que analizamos será una variable libre dentro de nuestra fórmula. Como esta variable se encuentra dentro de nuestra postcondición, siempre tendremos un modelo trivial que es el modelo en el que a esta variable se le asigna el valor *val*. Para solucionar esto tenemos dos opciones:

- Eliminar bof_i para la función inicial.
- Colocar un valor concreto de memoria en la dirección de memoria en donde debería estar la dirección de retorno.

Nuestro método hace uso de la segunda opción para solucionar el problema. Notemos que el valor concreto de memoria utilizado debe ser diferente del valor que usemos en la postcondición.

Otra cosa a tener en cuenta es que la postcondición propuesta nos permitirá detectar solo un buffer overflow en el programa. Esto se debe a que solo necesitamos que bof_i=val para algún valor de i para que toda la fórmula sea verdadera. Si deseamos obtener todos los buffer overflow del programa, es necesario realizar la verificación para cada una de las direcciones de retorno del programa de forma individual.

En conclusión, si existe un modelo del programa que tratamos de chequear, que haga que $wp.S.Q$ sea verdadera , entonces habremos encontrado un programa vulnerable a buffer overflow ya que esto implicaría que con los datos de entrada correctos somos capaces de sobrescribir una dirección de retorno con un valor arbitrario.

3.6. Chequeo de buffer overflow automatizado

Para automatizar el proceso de chequeo de buffer overflow se desarrollo un módulo aparte que hace uso del módulo de chequeo de buffer overflow.

El funcionamiento de este módulo es el siguiente:

1. Obtiene el listado de funciones del programa.
2. Extrae el código intermedio para cada una de las funciones del programa.
3. Chequea la presencia de buffer overflow en cada una de las funciones del programa.

Capítulo 4

Casos de estudio

En este capítulo veremos los resultados de aplicación del método desarrollado. Comenzaremos mostrando casos de prueba que muestran el correcto funcionamiento del método al analizar funciones que contienen llamadas interprocedurales. Luego veremos los casos de prueba desarrollados para probar la detección de buffer overflow en programas simples desarrollados por nosotros. Finalmente analizaremos los resultados de nuestro método aplicado a programas no sintéticos.

Los pasos a seguir para verificar los casos de estudio son:

1. Compilar el código, si tenemos el fuente.

2. Determinar la dirección dentro del binario de la función main. Para obtener la dirección de inicio de esta función podemos utilizar *get_functions*.

3. Obtener el código intermedio de esta función en formato json utilizando la dirección de la función main obtenida en 2. Para realizar esto podemos correr el siguiente comando: *toil -binrecurseat main main_address -tojson -o main.json*

4. Utilizar el programa desarrollado para transformar el código intermedio corriendo el siguiente comando: *python check_buffer_overflow.py -m main_address -j main.json -r main -o main.il.* Además de obtener el código intermedio transformado en el archivo main.il, el programa imprimirá en pantalla la postcondición que debemos utilizar en el último paso y las direcciones de memoria de donde tomarán los datos las funciones importadas.

5. Si el programa a analizar no tiene ciclos saltear este paso. Caso contrario, eliminar y expandir los ciclos del programa. El comando será el siguiente: *iltrans -il main.il -to-cfg -rm-indirect-ast -prune-cfg -unroll*

N -rm-cycles -to-ast -pp-ast main_done.il. La cantidad de veces que expandimos los ciclos depende de cada caso. Una buena opción es comenzar con un valor de N pequeño e ir incrementándolo.

6. Finalmente realizaremos la verificación usando topredicate. Junto con la postcondición que obtuvimos en el paso 4 corremos el comando: *topredicate -il main_done.il -post POSTCONDITION -solve -stp-out main.f.* Finalmente resolvemos la fórmula utilizando z3 con el comando: *z3 -smt2 main.f.* Esto nos devolverá un modelo del programa que verifica que la postcondición es válida luego de ejecutarse el programa o bien nos dirá que la fórmula es insatisfacible, por lo que no existe tal modelo y por lo tanto es imposible que esa postcondición se cumpla.

Para todos estos casos de prueba, también se mostrarán los resultados de la automatización del método.

4.1. Casos de prueba con llamadas interprocedurales.

A continuación veremos el resultado de aplicar nuestro método a programas simples donde se puede observar el comportamiento para llamadas interprocedurales. La verificación para todos estos casos se realizará utilizando postcondiciones y el comando topredicate.

4.1.1. Llamado a función.

En la Figura 4.1 podemos observar el caso de prueba más simple para probar el llamado entre funciones. Tenemos definida una función resta que es llamada desde la función main. Notemos de la línea 9 que la variable num está declarada pero no se le asigno ningún valor, por lo que esta variable será una variable libre dentro de la fórmula de la precondición más débil generada para este programa. Una tarea de verificación simple que nos gustaría realizar sobre este programa es determinar: dado un valor arbitrario (val), que valor debería tener la variable num para que luego de la ejecución del programa $result = val$.

Si recordamos de 2.1.1.1 en la página 10, el resultado de la función main será almacenado en el registro EAX, por lo que una postcondición válida para verificar lo que propusimos será $R\_EAX : u32 == val$. Recordemos que la postcondición debe ser anotada como si fuera una instrucción del lenguaje intermedio. Tomemos por ejemplo $val = 10$, luego la postcondición será $R\_EAX : u32 == 10 : u32$.

Recordemos que el código ensamblador de la función resta termina con una instrucción de tipo RET. Como BAP no soporta llamados entre funciones, si tratamos de correr este programa mediante la utilidad topredicate obtendremos un error. Por lo cual fue necesario aplicar el método desarrollado en el capítulo 3.

Luego de correr el método desarrollado sobre el programa, el código intermedio resultante será apto para ser utilizado con topredicate. Para corroborar que el resultado es apto, simplemente debemos llamar a topredicate con el código intermedio procesado de nuestro programa y la postcondición antedicha. Si todo funciona correctamente deberíamos obtener un modelo en el que el valor de la variable num sea 15.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int resta(int a, int b){
5          return a - b;
6  }
7
8  int main(int argc, char *argv[]) {
9          int num;
10         int result = resta(num, 5);
11         return result;
12 }
```

Figura 4.1: Función con llamada a otra función.

4.1.1.1. Resultado

```
1  sat
2  (model
3    (define-fun mem_56_array_101 () (Array (_ BitVec 32) (_ BitVec 8))
4      (_ as-array k!0))
5    (define-fun k!0 ((x!1 (_ BitVec 32))) (_ BitVec 8)
6      (ite (= x!1 #x7ffffff4) #x0f
7      (ite (= x!1 #x7ffffff5) #x00
8      (ite (= x!1 #x7ffffff6) #x00
9      (ite (= x!1 #x7ffffff7) #x00
10       #x00)))))
11 )
```

Figura 4.2: Modelo generado para simple.

En la Figura 4.2, podemos observar el modelo obtenido luego de resolver la fórmula generada de la precondición más débil utilizando el solver z3.

Podemos ver que en la línea 3 se define un array de vectores de 8 bits que tendrá como índice vectores de 32 bits y será nombrado k!0. Este arreglo representará la memoria de nuestro programa. Los vectores de 32 bits serán las direcciones de memoria de nuestro programa y los vectores de 8 bits serán los valores almacenados en esas direcciones. La memoria de nuestro programa será representada mediante la función de la línea 5. Esta función mapea a las direcciones de memoria 0x7ffffff4, 0x7ffffff5, 0x7ffffff6, 0x7ffffff7 los valores 0xf, 0x00, 0x00 y 0x00 respectivamente y al resto de las direcciones de memoria les asigna el valor 0x00 [1].

Si analizamos el código intermedio generado por el programa podemos llegar a la conclusión de que las direcciones de memoria antedichas son las que almacenan el valor del entero num de nuestro programa original. Son 4 direcciones ya que un entero en una arquitectura de 32 bits ocupa 32 bits, y cada dirección de memoria puede almacenar 8 bits. Notemos que el valor almacenado en la memoria es la representación hexadecimal en little endian de 15, que es el valor que esperábamos obtener.

4.1.2. Doble llamado a función.

Este caso fue creado para testear que el copiado de funciones funcionara correctamente. Podemos observar el código en la Figura 4.3. Al tener dos llamados a la misma función, la instrucción RET de la función suma puede realizar un salto indirecto hacia dos direcciones distintas como vimos anteriormente en 3.3. El código intermedio resultante debe contener dos copias de la función suma (que recibirán distintos nombres) con sus saltos indirectos respectivos arreglados para preservar la semántica del programa.

Nuevamente tenemos la variable entera num como variable libre. Para la verificación utilizaremos como postcondición $R\_EAX : u32 == 10 : u32$. Del código del programa podemos observar que para que el resultado de main sea 10, num debe valer 3.

4.1.2.1. Resultado

En la Figura 4.4 podemos ver el modelo resultante de haber corrido la utilidad topredicate con postcondición $R\_EAX : u32 == 10 : u32$ sobre el programa ya procesado por nuestro método. Como en el caso anterior, tenemos declarados un array y una función que representaran a la memoria.

[1] Para una comprensión más detallada del resultado ver [8]

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

int suma(int a, int b){
    return a + b;
}

int main(int argc, char *argv[]) {
    int num;
    int result = suma(num,5);
    int result2 = suma(result,2);
    return result2;
}
```

Figura 4.3: Función con dos llamadas a la misma función.

```
sat
(model
  (define-fun mem_56_array_103 () (Array (_ BitVec 32) (_ BitVec 8))
    (_ as-array k!0))
  (define-fun k!0 ((x!1 (_ BitVec 32))) (_ BitVec 8)
    (ite (= x!1 #x7ffffff0) #x03
    (ite (= x!1 #x7ffffff1) #x00
    (ite (= x!1 #x7ffffff2) #x00
    (ite (= x!1 #x7ffffff3) #x00
      #x00)))))
)
```

Figura 4.4: Modelo generado para double.

En este programa la variable entera num se encuentra entre las direcciones 0x7fffffff0 y 0x7fffff3. El valor almacenado es la representación hexadecimal de 3 en little endian que es el valor que esperábamos obtener.

4.1.3. Llamado anidado de funciones.

Este ejemplo fue creado para probar que el "copiado recursivo" de funciones, es decir realizar copias de funciones que se encuentran dentro de otras funciones, funcionara correctamente. Podemos observar el código del programa en la Figura 4.5. El código intermedio resultante debe contener copias de todas las funciones definidas y ser equivalente semánticamente al programa original. El chequeo se realiza, nuevamente, utilizando topredicate con la postcondición: $R\_EAX : u32 == 10 : u32$. El modelo esperado para este programa y esta postcondición es un modelo en el que $x = 8$.

```
#include <stdio.h>
#include <stdlib.h>

int suma(int a, int b){
    return resta(a, -1 * b)
}

int resta(int a, int b){
    return a - b;
}

int main(int argc, char *argv[]) {
    int x;
    int result = suma(x,2);
    return result;
}
```

Figura 4.5: Función con llamadas anidadas.

4.1.3.1. Resultado

En la Figura 4.6 podemos observar el modelo generado para este caso de prueba. En este caso la variable entera x se encuentra entre las direcciones 0x7fffffe8 y 0x7fffffeb. Como esperábamos, el valor asignado a x es 8.

```
sat
(model
  (define-fun mem_56_array_107 () (Array (_ BitVec 32) (_ BitVec 8))
    (_ as-array k!0))
  (define-fun k!0 ((x!1 (_ BitVec 32))) (_ BitVec 8)
    (ite (= x!1 #x7ffffe8) #x08
    (ite (= x!1 #x7ffffea) #x00
    (ite (= x!1 #x7ffffe9) #x00
    (ite (= x!1 #x7ffffeb) #x00
      #x00)))))
)
```

Figura 4.6: Modelo generado para anidado.

4.1.4. Llamado a función dentro de bucle.

Este ejemplo fue desarrollado para probar que el unrolling de las copias de las funciones funcionase correctamente. Podemos ver el código en la Figura 4.7. Nuevamente se prueba el correcto funcionamiento mediante la utilidad topredicate, utilizando como postcondición $R\_EAX : u32 == 10 : u32$.

Notemos que como en este ejemplo tenemos un bucle, será necesario realizar el unrolling de los ciclos del programa antes de poder llevar a cabo la tarea de verificación. La cantidad de veces que realicemos la expansión será determinante en el resultado de la verificación.

```
#include <stdio.h>
#include <stdlib.h>

int suma(int a, int b){
    return a + b;
}

int main(int argc, char *argv[]) {
    int x;
    int total = 0;
    int i = 0;
    for(i=0;i<2;i++){
        total = total + suma(x,2);
    }
    return total;
}
```

Figura 4.7: Función con llamada a otra función en un bucle.

4.1.4.1. Resultado

```
sat
(model
  (define-fun mem_56_array_106 () (Array (_ BitVec 32) (_ BitVec 8))
    (_ as-array k!0))
  (define-fun k!0 ((x!1 (_ BitVec 32))) (_ BitVec 8)
    (ite (= x!1 #x7ffffffa) #x00
    (ite (= x!1 #x7ffffffb) #x00
    (ite (= x!1 #x7ffffff8) #x03
    (ite (= x!1 #x7ffffff9) #x00
      #x00)))))
)
```

Figura 4.8: Modelo generado para función dentro de bucle.

En la Figura 4.8 podemos observar el modelo generado para este caso de prueba. En este caso la variable entera x se encuentra entre las direcciones 0x7fffffe8 y 0x7fffffeb. Como esperábamos, el valor asignado a x es 3.

Vale la pena notar que si utilizamos un valor impar como postcondición, o si realizamos una cantidad menor de 2 unroll de los loops del programa, obtendremos una fórmula que es insatisfacible.

4.1.5. Función recursiva.

La finalidad de este caso de prueba es simplemente probar que la detección de funciones recursivas funcione correctamente, ya que recordemos que el método desarrollado no soporta programas que tengan funciones recursivas. El código podemos observarlo en la Figura 4.9.

4.1.5.1. Resultado

En la Figura 4.10 podemos observar el error obtenido al tratar de aplicar el método a un programa recursivo. El error incluye el call stack de la función recursiva llamada.

4.2. Casos de prueba de funciones importadas

En esta sección no se llevara a cabo ninguna tarea de verificación, solo mostraremos los casos de prueba utilizados para probar el correcto funcio-

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

int suma(int a){
    if (a==0){
        return 0;
    }
    else{
        return a + suma(a-1);
    }
}

int main(int argc, char *argv[]) {
    int num;
    num = suma(4);
}
```

Figura 4.9: Función recursiva.

```
[*] Recursive call detected, we don't support recursive calls
[*] Printing call stack
[*]  0x8048404
[*]      0x80483dc
[*]              0x80483dc
```

Figura 4.10: Error obtenido al tratar de analizar una función recursiva.

namiento del método desarrollado con respecto al remplazo de las funciones importadas por sus equivalentes. Además de ver que el método funciona correctamente testearemos que la semántica de las funciones importadas desarrolladas por nosotros es similar. El testeo para estos casos se realizará mediante la evaluación del programa utilizando ileval mediante el comando: *ileval -il ejemplo.il -eval*.

4.2.1. Read

El ejemplo de la Figura 4.11 fue desarrollado para probar el correcto funcionamiento de la función read. Recordemos de 3.4 que los descriptores de archivo son remplazados por pedazos de memoria consecutivos de tamaño fijo. Al correr el método desarrollado se imprimirá en pantalla las dirección que utiliza cada una de las funciones importadas remplazadas para obtener sus datos de entrada. Por ejemplo:

```
[*] read_fun3_pc_0x1c is taking data from 0x80001000
```

Este mensaje nos indica que los datos de entrada para la función read provendrán de la dirección de memoria 0x80001000. Como a la hora de evaluar el programa no habrán datos en esta dirección de memoria, fue necesario agregar las siguientes instrucciones del lenguaje intermedio al programa modificado para facilitar la tarea de testeo.

```
mem:?u32 = mem:?u32 with [0x80001000:u32, e_little]:u32 = 0x41414141:u32
mem:?u32 = mem:?u32 with [0x80001004:u32, e_little]:u32 = 0x42424242:u32
```

Lo que hacen estas instrucciones es definir un valor para la memoria en el rango de direcciones 0x80001000 a 0x80001008 para que el programa tenga datos concretos sobre los que operar.

4.2.1.1. Resultado

En la Figura 4.12 podemos observar el resultado de la ejecución del caso de prueba. El resultado de la ejecución de ileval es el estado final de todos los registros y variables utilizadas por el programa junto con el estado de la memoria. En la Figura 4.12 omitimos el estado de las variables y registros y solamente nos concentramos en el estado de la memoria. Podemos observar entre las líneas 23 a 34 los datos de entrada de nuestro programa. Si analizamos el código ensamblador del programa veremos que la variable buffer de nuestro caso de prueba comienza en la dirección de memoria 0x7fffffe4 y va hasta la dirección 0x7fffffeb. Como podemos observar en el resultado de la ejecución, entre las líneas 12 y 19 tenemos el estado de la memoria en las direcciones antedichas y podemos corroborar que los valores son los esperados teniendo en cuenta los datos de entrada.

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
        char buffer[8];
        int fd = 0;
        fd = open("test.txt",O_RDONLY);
        int count = read(fd,buffer,8);
        return 0;
}
```

Figura 4.11: Caso de prueba para la función importada read.

4.2.2. Recv

Este ejemplo fue desarrollado para probar el correcto funcionamiento de la función importada recv. En la Figura 4.13 podemos apreciar el código utilizado. Al igual que en el caso anterior, los datos de entrada fueron tomados de la dirección de memoria 0x80001000 y fue necesario agregar instrucciones al código modificado para facilitar la tarea de testeo. Las instrucciones agregadas fueron, nuevamente:

```
mem:?u32 = mem:?u32 with [0x80001000:u32, e_little]:u32 = 0x41414141:u32
mem:?u32 = mem:?u32 with [0x80001004:u32, e_little]:u32 = 0x42424242:u32
```

Notemos, de la línea 11 de la Figura 4.13 que el tamaño de los datos de entrada es 8 pero solamente debemos recibir 6 de esos bytes.

4.2.2.1. Resultado

En la Figura 4.14 podemos observar el resultado de la ejecución del caso de prueba. Notemos que entre las líneas 21 a 28 se encuentran los datos de entrada, entre las direcciones 0x80001000 y 0x80001007. Y, que entre las líneas 12 a 17, se encuentra el estado de la memoria entre las direcciones 0x7fffffe6 a 0x7fffffeb, si analizamos el código ensamblador del programa notaremos que en esas direcciones se encuentra la variable buffer. Observemos que solo se copiaron 6 de los 8 bytes de los datos de entrada.

```
1  contents of variables
2  [...]
3  source_address_82_106:u32 = 0x80001000:u32
4  [...]
5  contents of memories
6  memory mem_56
7  [...]
8  7ffffffe0 -> 0:u8
9  7ffffffe1 -> 0:u8
10 7ffffffe2 -> 0:u8
11 7ffffffe3 -> 0:u8
12 7ffffffe4 -> 0x41:u8
13 7ffffffe5 -> 0x41:u8
14 7ffffffe6 -> 0x41:u8
15 7ffffffe7 -> 0x41:u8
16 7ffffffe8 -> 0x42:u8
17 7ffffffe9 -> 0x42:u8
18 7ffffffea -> 0x42:u8
19 7ffffffeb -> 0x42:u8
20 7ffffffec -> 0:u8
21 7ffffffed -> 0:u8
22 [...]
23 80001000 -> 0x41:u8
24 80001001 -> 0x41:u8
25 80001002 -> 0x41:u8
26 80001003 -> 0x41:u8
27 80001004 -> 0x42:u8
28 80001005 -> 0x42:u8
29 80001006 -> 0x42:u8
30 80001007 -> 0x42:u8
31 80001008 -> 0x43:u8
32 80001009 -> 0x43:u8
33 8000100a -> 0x43:u8
34 8000100b -> 0x43:u8
35 result: true
```

Figura 4.12: Resultado de ileval sobre el caso de prueba de read.

```
1  #include <stdio.h>
2  #include <errno.h>
3  #include <string.h>
4  #include <sys/types.h>
5  #include <sys/socket.h>
6
7  int main(int argc, char *argv[])
8  {
9          char buffer[6];
10         int sock = 0;
11         int count = recv(sock, buffer, 6, 0);
12         return 0;
13 }
```

Figura 4.13: Caso de prueba para la función importada recv.

```
1  contents of variables
2  [...]
3  source_address_82_105:u32 = 0x80001000:u32
4  [...]
5  contents of memories
6  memory mem_56
7  [...]
8  7ffffffe0 -> 0:u8
9  7ffffffe1 -> 0:u8
10 7ffffffe2 -> 0:u8
11 7ffffffe3 -> 0:u8
12 7ffffffe6 -> 0x41:u8
13 7ffffffe7 -> 0x41:u8
14 7ffffffe8 -> 0x41:u8
15 7ffffffe9 -> 0x41:u8
16 7ffffffea -> 0x42:u8
17 7ffffffeb -> 0x42:u8
18 7ffffffec -> 0x0:u8
19 [...]
20 7fffffff -> 0:u8
21 80001000 -> 0x41:u8
22 80001001 -> 0x41:u8
23 80001002 -> 0x41:u8
24 80001003 -> 0x41:u8
25 80001004 -> 0x42:u8
26 80001005 -> 0x42:u8
27 80001006 -> 0x42:u8
28 80001007 -> 0x42:u8
29 result: true
```

Figura 4.14: Resultado de ileval sobre el caso de prueba de recv.

4.2.3. Llamado compuesto

Este ejemplo fue desarrollado para probar que el método desarrollado funcione correctamente en programas que tienen más de una función importada. Esto quiere decir que ambas funciones deben ser remplazadas por su equivalente, cada función debe tomar datos de una dirección de memoria distinta y el programa en su conjunto debe hacer lo esperado. En la Figura 4.15 podemos observar el código del programa. Las funciones importadas utilizadas fueron read, y recv.

Los datos de entrada de la función read son tomados de la dirección de memoria 0x80001000 y los de la función recv de la dirección 0x80002000.

Al igual que en los casos anteriores se agregaron las siguientes instrucciones al código modificado.

```
mem:?u32 = mem:?u32 with [0x80001000:u32, e_little]:u32 = 0x41414141:u32
mem:?u32 = mem:?u32 with [0x80002000:u32, e_little]:u32 = 0x42424242:u32
```

Notemos que estamos escribiendo valores concretos para los datos de entrada de cada función.

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
        char buffer[9];
        int fd = 0;
        int sock = 0;
        int count = 0;
        fd = open("test.txt",O_RDONLY);
        count = read(fd,buffer,4);
        count = recv(sock, buffer+4, 4, 0);
        return 0;
}
```

Figura 4.15: Caso de prueba con varias funciones importadas.

4.2.3.1. Resultado

En la Figura 4.16 podemos observar el resultado de la ejecución del caso de prueba. Entre las líneas 14 a 17 podemos ver los datos de entrada para la

función read y entre las líneas 18 a 21 los datos de entrada de la función recv. Notemos que los datos comienzan en las direcciones de memoria 0x80001000 y 0x80002000 para cada una de las respectivas funciones. Entre las líneas 4 y 11 podemos observar que los datos han sido efectivamente copiados al buffer de 9 bytes.

```
1  contents of memories
2  memory mem_56
3  [...]
4  7ffffffe3 -> 0x41:u8
5  7ffffffe4 -> 0x41:u8
6  7ffffffe5 -> 0x41:u8
7  7ffffffe6 -> 0x41:u8
8  7ffffffe7 -> 0x42:u8
9  7ffffffe8 -> 0x42:u8
10 7ffffffe9 -> 0x42:u8
11 7ffffffea -> 0x42:u8
12 7ffffffeb -> 0:u8
13 [...]
14 80001000 -> 0x41:u8
15 80001001 -> 0x41:u8
16 80001002 -> 0x41:u8
17 80001003 -> 0x41:u8
18 80002000 -> 0x42:u8
19 80002001 -> 0x42:u8
20 80002002 -> 0x42:u8
21 80002003 -> 0x42:u8
22 result: true
```

Figura 4.16: Resultado de la ejecución del caso de prueba.

4.2.4. Strcpy

Este caso de prueba fue desarrollado para probar el correcto funcionamiento de la función strcpy. En la Figura 4.17 podemos observar el código desarrollado. Notemos que tenemos definidas 3 variables de tipo string: str, str2 y buffer. Las variables str y str2 fueron utilizadas para que al analizar el estado de la memoria, el efectivo copiado de los datos fuera más evidente. Si el código es ejecutado correctamente deberíamos ver primero las 4 A's, después las 4 B's, y finalmente las 4 C's.

```
#include <stdio.h>

int main(){
    char str[] = "AAAA";
    char buffer[8];
    char str2[] = "CCCC";
    strcpy(buffer, "BBBB");
    printf("%s\n", buffer);
}
```

Figura 4.17: Caso de prueba de la función importada strcpy.

4.2.4.1. Resultado

Antes de analizar el resultado tengamos en cuenta los valores hexadecimales para cada uno de los caracteres copiados en memoria. Los valores para las letras A, B y C son 0x41, 0x42 y 0x43 respectivamente.

En la Figura 4.18 podemos observar que el estado de la memoria es el esperado con la excepción que el orden de las letras se encuentra invertido. Recordemos de la sección 2.2 que el stack crece hacia direcciones inferiores de memoria lo que produce la alteración en el orden de las letras.

4.2.5. Memcpy

Este ejemplo fue desarrollado para probar el funcionamiento de la función importada memcpy. Como en el caso anterior, tenemos definidas dos variables de tipo string auxiliares: str y str2 que nos permitirán distinguir de forma más simple el copiado de los datos en memoria.

4.2.5.1. Resultado

En la Figura 4.20 podemos observar los resultados de la ejecución del caso de prueba. Notemos que entre las líneas 20 a 29 podemos ver la memoria de la variable str, entre las líneas 14 a 18 la memoria de la variable str2 y entre las líneas 8 a 13 podemos ver la variable buffer. Observemos que luego de las C's (0x43) se encuentren las 6 A's que fueron copiadas utilizando strcpy.

```
contents of variables
[...]
contents of memories
memory mem_56
[...]
7fffffde -> 0x43:u8
7fffffdf -> 0x43:u8
7fffffe0 -> 0x43:u8
7fffffe1 -> 0x43:u8
7fffffe2 -> 0:u8
7fffffe3 -> 0x42:u8
7fffffe4 -> 0x42:u8
7fffffe5 -> 0x42:u8
7fffffe6 -> 0x42:u8
7fffffe7 -> 0:u8
7fffffe8 -> 0:u8
7fffffe9 -> 0:u8
7fffffea -> 0:u8
7fffffeb -> 0x41:u8
7fffffec -> 0x41:u8
7fffffed -> 0x41:u8
7fffffee -> 0x41:u8
7fffffef -> 0:u8
result: true
```

Figura 4.18: Resultado de ileval sobre el programa de prueba de strcpy.

```
#include <stdio.h>

int main(){
    char str[] = "AAAAAABBBB";
    char str2[] = "CCCCC";
    char buffer[8];
    memcpy(buffer, str, 6);
    printf("%s\n", buffer);
}
```

Figura 4.19: Caso de prueba para la función importada memcpy.

```
1  contents of variables
2  [...]
3  contents of memories
4  memory mem_56
5  [...]
6  7fffffca -> 0:u8
7  7fffffcb -> 0:u8
8  7fffffd7 -> 0x41:u8
9  7fffffd8 -> 0x41:u8
10 7fffffd9 -> 0x41:u8
11 7fffffda -> 0x41:u8
12 7fffffdb -> 0x41:u8
13 7fffffdc -> 0x41:u8
14 7fffffdf -> 0x43:u8
15 7ffffe0 -> 0x43:u8
16 7ffffe1 -> 0x43:u8
17 7ffffe2 -> 0x43:u8
18 7ffffe3 -> 0x43:u8
19 7ffffe4 -> 0:u8
20 7ffffe5 -> 0x41:u8
21 7ffffe6 -> 0x41:u8
22 7ffffe7 -> 0x41:u8
23 7ffffe8 -> 0x41:u8
24 7ffffe9 -> 0x41:u8
25 7ffffea -> 0x41:u8
26 7ffffeb -> 0x42:u8
27 7ffffec -> 0x42:u8
28 7ffffed -> 0x42:u8
29 7ffffee -> 0x42:u8
30 7fffffef -> 0:u8
```

Figura 4.20: Resultado de ileval en el caso de prueba de memcpy.

4.3. Casos de prueba de buffer overflow

4.3.1. Read, recv

Para probar el correcto funcionamiento del código desarrollado se utilizaron algunos ejemplos simples de buffer overflow. En la Figura 4.21 podemos observar un programa que lee datos de un archivo `test.txt` y copia los primeros 28 bytes de ese archivo a un buffer ubicado en el stack. El problema surge porque el buffer en el stack puede almacenar solo 4 bytes, por lo que los 24 bytes restantes sobrescribieran los datos adyacentes en la memoria.

```
int main(int argc, char *argv[]) {
        char buffer[4];
        int fd = 0;
        fd = open("test.txt",O_RDONLY);
        int count = read(fd,buffer,28);
        return 0;
}
```

Figura 4.21: Buffer overflow con read.

En la Figura 4.22 podemos observar un caso similar que utiliza sockets en vez de archivos. Recordamos de la sección 3.4 que los descriptores de archivos devueltos por las funciones read y recv serán remplazados por pedazos de memoria que harán de variables libres en la fórmula de la precondición más débil de estos programas.

```
int main(int argc, char *argv[]) {
        char buffer[4];
        int sock = 0;
        int count = recv(sock, buffer, 28, 0);
        return 0;
}
```

Figura 4.22: Buffer overflow con recv.

Los datos de entrada de la función read son tomados de la dirección de memoria 0x80001000. Y la postcondición utilizada fue:

```
(bof_6_10:u32 == 0xcafecafe:u32)|
(bof_7_10:u32 == 0xcafecafe:u32)
```

4.3.1.1. Resultados

En la Figura 4.23 podemos observar el modelo generado para este caso de prueba. Antes que nada cabe aclarar que el hecho de que exista un modelo que satisface la postcondición dada es indicativo de que existe buffer overflow en el programa. Ahora, si analizamos el modelo obtenido podemos concluir que alguna de las direcciones de retorno del programa (bof_i) tendrá el valor 0xcafecafe solo si el estado de la memoria es el siguiente:

- memoria[0x8000101c] = 0xfe
- memoria[0x8000101d] = 0xca
- memoria[0x8000101e] = 0xfe
- memoria[0x8000101f] = 0xca

```
sat
(model
  (define-fun mem_56_array_110 () (Array (_ BitVec 32) (_ BitVec 8))
    (_ as-array k!0))
  (define-fun k!0 ((x!1 (_ BitVec 32))) (_ BitVec 8)
    (ite (= x!1 #x8000101e) #xfe
    (ite (= x!1 #x8000101f) #xca
    (ite (= x!1 #x8000101c) #xfe
    (ite (= x!1 #x8000101d) #xca
      #xfe)))))
)
```

Figura 4.23: Modelo del programa bof_read que prueba la existencia de buffer overflow.

Recordemos que los datos de entrada del programa se encuentran en la dirección 0x80001000, lo que indica que estamos sobrescribiendo la dirección de retorno con datos controlados por el usuario. Además, los datos que sobrescriben a la dirección son los que están entre los offset 0x1c a 0x1f.

Notemos que la verificación no solo nos permite detectar si la función es vulnerable, sino que también nos permite saber el offset exacto de los datos de entrada que debemos controlar para poder tomar control del programa.

4.3.2. Strcpy, memcpy

Este caso de prueba fue desarrollado para verificar el correcto funcionamiento de la detección de buffer overflow. Este caso de prueba es de una

complejidad mayor que el visto anteriormente ya que además de leer los datos, realiza el copiado de los mismos mediante strcpy dentro de una función auxiliar.

Los datos de entrada de la función read son tomados de la dirección de memoria 0x80001000. Y la postcondición utilizada fue:

```
(bof_6_10:u32 == 0xcafecafe:u32) |
(bof_10_10:u32 == 0xcafecafe:u32 |
 bof_11_10:u32 == 0xcafecafe:u32 |
 bof_12_10:u32 == 0xcafecafe:u32)
```

En la Figura 4.24 podemos observar el código del caso de prueba.

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
        char buffer[35];
        int fd = 0;
        fd = open("test.txt",O_RDONLY);
        int count = read(fd,buffer,35);
        vulnerable(buffer);
        return 0;
}

int vulnerable(char *str){
    char bof[4];
    strcpy(bof, str);
    return 0;
}
```

Figura 4.24: Caso de prueba de buffer overflow con strcpy.

4.3.2.1. Resultados

En la Figura 4.25 podemos observar el modelo que prueba la existencia de buffer overflow en el programa. Notemos que los bytes que sobrescriben la dirección de retorno son los bytes 0x10, 0x11, 0x12 y 0x13 de los datos de entrada.

```
sat
(model
  [...]
  (define-fun k!0 ((x!1 (_ BitVec 32))) (_ BitVec 8)
    [...]
    (ite (= x!1 #x80001010) #xfe
    (ite (= x!1 #x80001011) #xca
    (ite (= x!1 #x80001012) #xfe
    (ite (= x!1 #x80001013) #xca
    [...]
    (ite (= x!1 #x7ffffffa) #x00
      #x01))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
)
```

Figura 4.25: Modelo del programa que prueba la existencia de buffer overflow.

4.3.3. ActFax Server Buffer Overflow

ActiveFax Server 4.3.2 es una aplicación de 32 bits que corre en Windows que permite el envío y recepción de fax a través de la red. Es posible importar nuevos usuarios desde un archivo con formato csv. Debido a la falta de chequeo en el tamaño del nombre de usuario, es posible copiar una cantidad de datos arbitrarios en el stack produciéndose así un buffer overflow (OSVDB-ID 85175).

4.3.3.1. Resultado

La parte más importante del código de la función vulnerable podemos observarlo en la Figura 4.26. El buffer overflow se produce por el llamado a la función lstrcpyA de la línea 10. En el ret de la última línea se produce la transferencia de control hacia una dirección controlada por el usuario.

Como podemos observar en la línea 12 de la Figura 4.26, la función 0x518CE0 realiza un llamado a la función 0x4862E0 (sub_4862E0). Si analizamos esta función por dentro veremos que existe más de una instrucción de tipo ret. El método desarrollado hace un análisis lineal de las funciones y

```
1  .text:00518CE0    sub     esp, 208h
2  .text:00518CE6    mov     eax, [esp+208h+lpString2]
3  .text:00518CED    push    ebx
4  .text:00518CEE    push    ebp
5  .text:00518CEF    push    esi
6  .text:00518CF0    push    edi
7  .text:00518CF1    lea     ecx, [esp+218h+String2]
8  .text:00518CF5    push    eax             ; lpString2
9  .text:00518CF6    push    ecx             ; lpString1
10 .text:00518CF7    call    ds:lstrcpyA
11 [...]
12 .text:00518D71    call    sub_4862E0
13 [...]
14 .text:00518DBB    pop     edi
15 .text:00518DBC    pop     esi
16 .text:00518DBD    pop     ebp
17 .text:00518DBE    pop     ebx
18 .text:00518DBF    add     esp, 208h
19 .text:00518DC5    retn
```

Figura 4.26: Función vulnerable en ActFax Server.

asume que la función termina al encontrar una instrucción de tipo ret. Es por esto que el código intermedio obtenido al aplicar el método desarrollado a la función 0x518CE0 no es válido para realizar verificaciones ya que no respeta la semántica original de la función.

4.3.4. Blaze Video HDTV Player

Blaze Video HDTV Player 6.6 es una aplicación de 32 bits que corre en Windows que sirve para reproducir archivos multimedia. Soporta el uso de playlists en formato plf. Debido a una falta de control en el tamaño de los datos que se copian al realizar el parseo del playlist es posible copiar una cantidad arbitraria de datos al stack produciéndose un buffer overflow (OSVDB-ID 80896).

El copiado de los datos se lleva a cabo en la función 0x6400f670 del módulo MediaPlayerCtrl.dll. El copiado es realizado por una instrucción rep movs como podemos observar en la línea 10 de la Figura 4.27.

4.3.4.1. Resultado

La función vulnerable en este programa es la 0x6400f670. Como podemos observar en la Figura 4.28, esta función realiza un llamado a la función

```
1  .text:6400F670     sub      esp, 218h
2  .text:6400F676     push     ebx
3  .text:6400F677     push     ebp
4  [...]
5  .text:6400F6E3     sub      edi, ecx
6  .text:6400F6E5     mov      eax, ecx
7  .text:6400F6E7     mov      esi, edi
8  .text:6400F6E9     mov      edi, [esp+228h+var_218]
9  .text:6400F6ED     shr      ecx, 2
10 .text:6400F6F0     rep movsd                ; strcpy
11 [...]
12 .text:6400F807     pop      esi
13 .text:6400F808     pop      ebp
14 .text:6400F809     xor      eax, eax
15 .text:6400F80B     pop      ebx
16 .text:6400F80C     add      esp, 218h
17 .text:6400F812     retn     10h
```

Figura 4.27: Función vulnerable en Blaze

0x64038ad3, esta función llama a subfunciones que terminan llamando nuevamente a 0x64038ad3.

```
[*] Recursive call detected, we don't support recursive calls
[*] Printing call stack
[*]  0x6400f670
[*]      0x6400fb80
[*]              0x64043e41
[*]                      0x64038ad3 (_malloc)
[*]                              0x64038ae5 (__nh_malloc)
[*]                                      0x64038b11
[*]
0x6403d132 (_lock)
[*]
0x64038ad3 (_malloc)
```

Figura 4.28: Llamado recursivo de funciones detectado.

4.3.5. Avaya WinPMD UniteHostRouter Buffer Overflow

Avaya WinPMD 3.8.2 es un aplicación de 32 bits que corre en Windows que sirve para configurar teléfonos Avaya IP DECT. El servicio UniteHostRouter, que utiliza conexiones UDP, realiza un copiado de datos al stack sin

tener en cuenta el tamaño de los mismos, lo que resulta en un buffer overflow (OSVDB-ID 73269, 82764).

4.3.5.1. Resultado

La función que realiza el copiado de los datos es la función 0x403160, el copiado de los datos se realiza mediante una instrucción rep movsb, sin embargo la dirección de retorno que se sobrescribe al producirse el buffer overflow no es la de esta función. La dirección de retorno que se pisa es la de la función 0x401F90, esta función es la que luego llama a 0x403160.

```
1  [...]
2  .text:00402020     jmp      ds:off_40247C[eax*4]
3  .text:00402027
4  .text:00402027 loc_402027:
5  .text:00402027     lea      eax, [esp+134h+hostshort]
6  .text:0040202B     push     eax                ; int
7  .text:0040202C     push     esi                ; Str
8  .text:0040202D     call     sub_4030B0
9  .text:00402032     add      esp, 8
10 .text:00402035     test     eax, eax
11 .text:00402037     jz       short loc_40206B
12 .text:00402039     mov      edx, dword_46BF48
13 .text:0040203F     mov      ecx, [esp+134h+hostshort]
14 .text:00402043     mov      eax, [esp+134h+len]
15 .text:0040204A     push     ecx                ; hostshort
16 .text:0040204B     mov      ecx, [esp+138h+buf]
17 .text:00402052     shl      edx, 5
18 .text:00402055     add      edx, offset unk_46BE00
19 .text:0040205B     push     edx                ; optval
20 .text:0040205C     push     eax                ; len
21 .text:0040205D     push     ecx                ; buf
22 .text:0040205E     call     sub_4024A0
23 .text:00402063     add      esp, 10h
24 .text:00402066     jmp      loc_40212F
25 .text:0040206B ; ---------------------------------------------------------------
26 .text:0040206B
27 .text:0040206B loc_40206B:
28 .text:0040206B     lea      edx, [esp+134h+ArgList]
29 .text:0040206F     push     edx                ; int
30 .text:00402070     push     esi                ; Str
31 .text:00402071     call     sub_403160
32 [...]
```

Figura 4.29: Código ensamblador de la función 0x401F90 de Avaya WinPMD

El llamado a 0x403160 se realiza a través del uso de saltos indirectos (jump table). Como topredicate no puede analizar correctamente esta función debido a los saltos indirectos, y nuestro método tampoco soporta saltos de este tipo, no es posible realizar la verificación. En la Figura 4.29 podemos ver un extracto del código de la función 0x401F90. En la segunda línea podemos observar el salto indirecto mediante una jump table, y en la penúltima línea podemos ver el llamado a la función 0x403160.

Notemos de las líneas 8, 12, y 20 de la Figura 4.30 que el llamado de funciones se realiza a través de saltos indirectos cargando la dirección de la función en el registro EDI. Además, la función tiene dos instrucciones de tipo RET.

4.3.6. Otros programas

Aquí nombraremos otros intentos de detección de buffer overflow que fallaron y detallaremos brevemente porque la verificación no fue posible.

- ASX to MP3 Converter 1.8 (Windows x86)[2].
- NIPrint (Windows x86 CVE-2003-1141).
- Savant Web Server 3.1 (Windows x86 CVE-2002-1120).
- Simple Web Server 2.2 (Windows x86 OSVDB-ID 84310).

En el caso de ASX to MP3 Converter y de NiPrint, el método falla debido a que en la función en donde se produce el buffer overflow contiene saltos indirectos. Con respecto a Savant la función vulnerable contiene múltiples direcciones de retorno. Finalmente, el método no funciona con Simple Web Server debido a que el buffer overflow se produce por el llamado a una función importada vsprintf que no se encuentra entre las funciones importadas que desarrollamos.

4.4. Resultados de la automatización

El chequeo automatizado fue corrido con todos los casos de prueba mencionados anteriormente. Los resultados del proceso pueden observarse en la Figura 4.31.

Los programas utilizados para probar los casos de prueba interprocedurales se encuentran agrupados en la misma categoría ya que las funciones que

[2]https://www.exploit-db.com/exploits/38382/

```
1  .text:00403160     push    ebx
2  .text:00403161     push    esi
3  .text:00403162     mov     esi, [esp+8+Str]
4  .text:00403166     push    edi
5  .text:00403167     mov     edi, ds:strpbrk
6  .text:0040316D     push    offset Control  ; "\n\r"
7  .text:00403172     push    esi             ; Str
8  .text:00403173     call    edi ; strpbrk
9  .text:00403175     push    offset asc_45E510 ; "/\n\r"
10 .text:0040317A     push    esi             ; Str
11 .text:0040317B     mov     ebx, eax
12 .text:0040317D     call    edi ; strpbrk
13 .text:0040317F     mov     esi, eax
14 .text:00403181     add     esp, 10h
15 .text:00403184     cmp     esi, ebx
16 .text:00403186     jnb     short loc_4031BD
17 .text:00403188     inc     esi
18 .text:00403189     push    offset a?_4     ; ":/? \n\r"
19 .text:0040318E     push    esi             ; Str
20 .text:0040318F     call    edi ; strpbrk
21 .text:00403191     add     esp, 8
22 .text:00403194     cmp     eax, ebx
23 .text:00403196     ja      short loc_4031BD
24 .text:00403198     mov     edx, [esp+0Ch+arg_4]
25 .text:0040319C     sub     eax, esi
26 .text:0040319E     mov     ecx, eax
27 .text:004031A0     mov     edi, edx
28 .text:004031A2     mov     ebx, ecx
29 .text:004031A4     shr     ecx, 2
30 .text:004031A7     rep movsd
31 .text:004031A9     mov     ecx, ebx
32 .text:004031AB     and     ecx, 3
33 .text:004031AE     rep movsb
34 .text:004031B0     pop     edi
35 .text:004031B1     mov     byte ptr [eax+edx], 0
36 .text:004031B5     pop     esi
37 .text:004031B6     mov     eax, 1
38 .text:004031BB     pop     ebx
39 .text:004031BC     retn
40 .text:004031BD ; ---------------------------------------
41 .text:004031BD
42 .text:004031BD loc_4031BD:
43 .text:004031BD
44 .text:004031BD     pop     edi
45 .text:004031BE     pop     esi
46 .text:004031BF     xor     eax, eax
47 .text:004031C1     pop     ebx
48 .text:004031C2     retn
```

Figura 4.30: Función en donde se realiza el copiado de Avaya WinPMD.

Programa	Func.	Func. vuln.	Errores	Falsos Positivos	Unsat
Casos de prueba interprocedurales	10/11	3	2	3	5/6
Caso Recursivo	-	-	-	-	-
Read/Recv (gcc)	10	4	2	3	4
Read+Strcpy	11	5	2	3	4
ActFax	-	-	-	-	-
Blaze Video HDTV Player	-	-	-	-	-
Avaya WINPMD	-	-	-	-	-

Figura 4.31: Tabla de resultados de la automatización

se detectan como vulnerables son funciones agregadas por el compilador en todos los casos. La función main de esos programas no es detectada como vulnerable en ningún caso.

Las funciones vulnerables de los casos de prueba interprocedurales son: deregister_tm_clones, register_tm_clones y _fini.

En el caso de las funciones register_tm_clones y deregister_tm_clones el falso positivo se debe a que hay varias instrucciones de tipo ret en el código, lo que produce que se creen varias variables auxiliares del tipo bof_i (recordemos que la postcondición es de la forma $bof_1 = val \vee bof_2 = val \vee \ldots \vee bof_n = val$). Sin embargo, la fórmula resultante no hace uso de todas las variables de tipo bof_i, entonces estas son consideradas como variables libres para el solver, por lo que el solver les asignará el valor val para hacer la fórmula satisfacible.

El falso positivo de la función _fini se debe a una incorrecta conversión del código binario del programa por parte de BAP.

Para los dos casos de prueba de buffer overflow tenemos los mismos 3 falsos positivos de los programa de prueba antes mencionados y además tenemos la función main, que es vulnerable a buffer overflow en ambos casos. Notemos que el caso de prueba que utiliza strcpy tiene una función vulnerable más que el otro caso. Esto se debe a que el parámetro str de la función

no está definido al momento de realizar el análisis por lo que el solver puede elegir una configuración de memoria arbitraria que produzca un buffer overflow.

Como podemos notar de la tabla de la Figura 4.31, las columnas de los programas Avaya WINPMD, ActFax y Blaze Video HDTV Player se encuentran vacías. Esto se debe a errores producidos en ambos programas que veremos a continuación. Al correr la utilidad toil en Avaya WINPMD para obtener el código intermedio del ejecutable de en formato JSON obtenemos un stack overflow. Si intentamos obtener el código intermedio de cada una de las funciones del programa obtenemos un error de librería faltante al correr get_functions. Con respecto a ActFax y Blaze, al correr toil obtenemos el siguiente error: Fatal error: exception Failure("bits2segreg: reserved"), y al utilizar get_functions obtenemos un stack overflow.

Capítulo 5

Conclusión y trabajos futuros

BAP es un framework de análisis binario bastante completo que cuenta con un lenguaje intermedio que facilita la tarea de análisis y verificación de código binario. Una carencia importante que tiene es la imposibilidad de realizar análisis de funciones interprocedurales, lo que limita en gran medida su utilidad.

En este trabajo se desarrollo una herramienta para verificar de manera estática la existencia de buffer overflow en código binario mediante el uso del framework BAP. La herramienta esta compuesta por un parser de BIL, dumper de BIL y un módulo que modifica el código intermedio para agregar soporte a la verificación. Además de usar el código intermedio utilizamos varias de las utilidades de BAP junto con un SMT Solver para llevar a cabo el proceso.

La herramienta desarrollada también puede ser utilizada para realizar otro tipo de tareas de verificación más generales gracias al soporte de llamados interprocedurales y sustitución de funciones importadas que desarrollamos.

El método desarrollado funcionó para programas simples como pudimos ver en los resultados, pero falló para los casos de prueba reales en los que se intentó realizar la verificación. Las razones de esto fueron la existencia de saltos o llamadas indirectas, la presencia de múltiples ret en una sola función, la existencia de funciones recursivas y la existencia de buffer overflow causados por funciones importadas no soportadas.

Una posible solución al problema de múltiples ret en una misma función seria dividir la función en subfunciones tales que cada una de ellas tenga solo un ret. Esto podría llevarse a cabo determinando los paths de la función que llegan a cada uno de sus ret y separando la función de acuerdo a esos paths.

Es posible que haya casos reales en los que el método desarrollado funcione correctamente pero para poder realizar la prueba es necesario encontrar un programa que sea vulnerable a buffer overflow basado en el stack para el

cual ya exista un exploit público. Es requerimiento que ya exista un exploit público para el programa. Es necesario, ya que encontrar un buffer overflow de manera manual en un programa es una tarea muy lenta; a partir de este exploit, determinar porque se produce el buffer overflow. En la gran mayoría de los casos, no existe documentación que explique la vulnerabilidad en el programa a explotar, por lo que es necesario realizar un análisis manual del binario junto con el exploit público para determinar porque sucede el buffer overflow y la función vulnerable. Finalmente, se puede proceder a realizar la verificación. El proceso antes mencionado es lento por lo que realizar un caso de prueba nuevo para el método desarrollado es un proceso que toma bastante tiempo.

De todas maneras creemos que nuestro trabajo tiene relevancia en dos aspectos. Por un lado se estudió y documento el framework BAP, además de extenderlo con software propio para poder llevar a cabo la tarea que propone esta tesina. Esto puede servir para futuros usos de la herramienta como así también para extender o modificar el presente trabajo. Por otro lado los resultados negativos de este trabajo muestran el desafío que propone la verificación formal estática de código binario. Los programas en este lenguaje poseen toda la complejidad del bajo nivel de instrucciones que posee, como la no estructuración de los bloques de programa, la manipulación explícita del stack, etc.

5.1. Trabajos futuros

A continuación mencionaremos algunas de las mejoras que pueden ser realizadas sobre este trabajo.

Algoritmo de extracción de funciones

El algoritmo utilizado para determinar donde termina una función es realmente sencillo, simplemente buscamos la primera instrucción de tipo RET y tomamos esa instrucción como la ultima instrucción de la función analizada. Como pudimos observar en los resultados, el uso de un algoritmo tan simple nos trae problemas ya que, por ejemplo, hay funciones que tienen más de una instrucción de retorno. Es por ello que consideramos que una mejora importante que podría realizarse es remplazar este algoritmo por uno más complejo que permita trabajar sobre un conjunto mayor de funciones extendiendo el alcance de la herramienta.

Funciones importadas

La cantidad de programas que podemos verificar esta directamente relacionado con la cantidad de funciones importadas que tengamos desarrolladas. Es por ello que consideramos que agregar soporte para más funciones nos permitiría verificar un abanico más grande de programas. En este trabajo desarrollamos algunas de las funciones que generalmente son las responsables de la existencia de buffer overflow como memcpy, strcpy junto con dos de las funciones de entrada de datos más importantes read y recv. Sin embargo, la lista de funciones que se recomienda no utilizar por ser conocidas causantes de buffer overflow es grande, algunos ejemplos de funciones que podrías agregarse son sprintf, gets, fgets, strcat. Lo mismo vale para las funciones de entrada de datos, un posible agregado serían las funciones de entrada de datos mediante diálogos en sistemas Windows.

Soporte para otras verificaciones

En este trabajo solo desarrollamos el soporte para la verificación de buffer overflow, pero variando la postcondición utilizada y agregando instrucciones en el código intermedio del programas es posible realizar otro tipo de verificaciones de seguridad, por ejemplo, detectar la existencia de integer overflow.

Bibliografía

[1] http://en.wikipedia.org/wiki/program_slicing.

[2] http://en.wikipedia.org/wiki/static_single_assignment_form.

[3] http://es.wikipedia.org/wiki/extensible_markup_language.

[4] http://es.wikipedia.org/wiki/extension_de_signo.

[5] http://json.org/json-es.html.

[6] http://llvm.org/docs/langref.html.

[7] https://github.com/google/protobuf/.

[8] http://smt-lib.org/.

[9] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[10] http://www.cs.nyu.edu/acsys/cvc3/.

[11] http://www.graphviz.org/.

[12] Ullman et. al's Aho. *Compilers Principles, Techniques and Tools Second Edition*. Addison Wesley, 2007.

[13] David Brumley. *Analysis and Defense of Vulnerabilities in Binary Code*. PhD thesis, CMU, 2008.

[14] Silvio Cesare Silvio Cesare and Yang Xiang. Wire – a formal intermediate language for binary analysis wire – a formal intermediate language for binary analysis wire – a formal intermediate language for binary analysis wire – a formal intermediate language for binary analysis. *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, 2012.

[15] Edward J. Schwartz David Brumley, Ivan Jager and Spencer Whitman. *The BAP Handbook*. Pagina 16, October 2013.

[16] Julio Auto de Medeiros. *Developing an intermediate representation for analyzing x86 binary programs*. PhD thesis, UFPE / CIN (Universidade Federale de PErnambuco, Centro de INformatica, Brazil), 2007.

[17] Edsger W. Dijkstra. Ewd472: Guarded commands, non-determinacy and formal. derivation of programs. August 2006.

[18] Thomas Dullien and Sebastian Porst (zynamics). Reil: A platform-independent intermediate representation reil: A platform-independent intermediate representation reil: A platform-independent intermediate representation reil: A platform-independent intermediate representation reil: A platform-independent intermediate representation of disassembled code for static code analysis. *CansecWest*, 2009.

[19] Daniel Kroening et Ofer Strichman. *Decision Procedures An Algorithmic Point of View*. Springer, 2008.

[20] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2014.

[21] Steven S. Muchnic. *Advanced Compiler Design And Implementation*. Morgan Kaufmann Publishers, 1997.

[22] Alejandro Ezequiel Orbe. *Symmetries in Automated Reasoning, The case of Modal Logics and Satisfiability Modulo Theories*. PhD thesis, 2014.