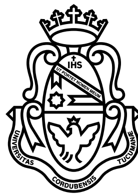
Un entorno de ejecución de C para detectar comportamientos indefinidos

Autores: Dionisio E. Alonso y Leandro S. Perona
Director: Lic. Daniel F. Moisset

Trabajo especial de grado

Un entorno de ejecución de C para detectar comportamientos indefinidos

Autores: Dionisio E. Alonso y
Leandro S. Perona
Director: Lic. Daniel F. Moisset



Universidad Nacional de Córdoba
Facultad de Matemática, Astronomía, Física y Computación

Córdoba, 18 de diciembre de 2015

Esta obra se distribuye bajo una licencia Creative Commons Atribución 4.0 Internacional

ABSTRACT

With the growing use of the *C* language, became evident the need to create a standard which guaranties that the same source code produces the same program when compiled with different compilers or on different platforms. However, said standard leaves a series of situations for which the behaviour of the program is undefined. Ignoring these behaviours leads to difficulties in assuring the program's portability and the consistency of errors.

The goal of this work is to present an environment that aids the software developer in identifying those situations in which a program written in *C* could produce an undefined behavior.

Classification: D.3.4 – Compilers; Debuggers; Run-time environments
Keywords: ANSI C; C99; ISO/IEC 9899:1999; Clang; Software Portability; Undefined Behavior

RESUMEN

Con el creciente uso del lenguaje *C*, se vio la necesidad de crear un estándar que garantice que el mismo código fuente produce el mismo programa, al compilarse con distintos compiladores o en distintas plataformas. No obstante, dicho estándar deja una serie de situaciones para las que el comportamiento del programa es indefinido. La ignorancia de estos comportamientos llevan a dificultades para garantizar la portabilidad y consistencia ante errores, de los programas.

El objetivo de este trabajo es presentar un entorno que ayude al desarrollador a detectar las situaciones en que una pieza de software escrita en el lenguaje *C* pueda tener un comportamiento indefinido.

Clasificación: D.3.4 – Compilers; Debuggers; Run-time environments
Palabras clave: ANSI C; C99; ISO/IEC 9899:1999; Clang; Portabilidad; Undefined Behavior

AGRADECIMIENTOS

A mis padres y hermana, por todos estos años de aguante y paciencia.
A mi abuela, por sus deudas adquiridas con santos previo a mis exámenes.
A Carolina, por toda la compañía y el apoyo incondicional en la etapa final de mi carrera.
A mis compañeros de la facultad, con los que pasara por tantas jornadas de estudio y noches de entrega.
A mi compañero Leandro, por esperarme que alcanzara los requisitos para presentar este trabajo.

Dionisio

A mis padres, por su apoyo y paciencia invariables durante tantos años de carrera, tanto buenos como malos.
A Cecilia, por todo su cariño y las fuerzas para seguir adelante hasta la culminación de este trabajo que me dio.
A todos los compañeros y profesores que me ayudaron a dar un paso más en esta carrera, y en particular a mi compañero Dionisio, con quien recorrí codo a codo los últimos años.

Leandro

A nuestro director Daniel, por la paciencia en el pausado desarrollo de nuestro trabajo de grado.
A Carolina, por los gráficos del informe y revisiones de ortografía y puntuación general; por sus comentarios y sugerencias.
Finalmente, un agradecimiento especial a Nicolás, Miguel, Daniel y Leonardo, por aceptar ser parte del tribunal evaluador de este trabajo.

Dionisio y Leandro

ÍNDICE GENERAL

1. Introducción **1**

2. Marco teórico **3**
2.1. El lenguaje C . . . 3
2.2. Estándar ANSI C99 . . . 3
2.3. Comportamiento Indefinido . . . 4
2.4. El lenguaje C++ . . . 5
2.5. LLVM . . . 5
2.5.1. LLVM Intermediate Representation . . . 5
2.6. Clang . . . 6
2.7. Abstract Syntax Tree . . . 6
2.8. Compilador “source to source” . . . 7
2.9. Instrumentación . . . 7

3. Realización **9**
3.1. Metodología de trabajo . . . 9
3.1.1. Comportamientos indefinidos y casos de prueba . . 9
3.1.2. Compilador y entorno de ejecución . . . 11
3.2. Primer enfoque: *libtooling* y *libastmatchers* . . . 12
3.3. Enfoque definitivo: *CodeGen* . . . 22
3.3.1. Primeros casos, `div` y `rem` . . . 24
3.3.2. Los operadores *Shift* derecho e izquierdo . . . 27
3.3.3. Información sobre objetos en memoria . . . 30
3.3.4. Información sobre arreglos multidimensionales . . . 40
3.3.5. Los punteros *Just-Beyond* . . . 50

4. El entorno de ejecución `idbre` **61**

5. Prueba de mundo real: htop **67**

6. Otras herramientas **71**
6.1. Comparación con el UndefinedBehaviorSanitizer de Clang . 71

7. Conclusiones **73**

8. Trabajo a futuro **75**

A. Listado de comportamientos resueltos **77**

B. Código fuente **81**

1 INTRODUCCIÓN

El lenguaje de programación *C* fue desarrollado entre los años 1969 y 1973 para su uso en el sistema operativo Unix [10]. Durante las décadas del '70 y '80 su uso se extendió significativamente y la diversidad de implementaciones existentes llevó a la creación de una especificación estándar del lenguaje, cuya última versión, a la que nos referiremos de aquí en adelante, es ISO/IEC 9899:1999 [8].

Hoy en día continúa siendo uno de los lenguajes más utilizados, y con él se implementan sistemas operativos usados en computadoras personales y sistemas críticos, aplicaciones de usuario y hasta otros lenguajes de programación.

Sin embargo, el estándar de *C* deja el comportamiento de muchas construcciones del lenguaje sin una única especificación, para las cuales cada compilador puede decidir el comportamiento.

Algunas de estas construcciones dependen de detalles de la implementación o de convenciones de localización. En estos casos, cada compilador debe documentar qué comportamiento les otorga. Para otras de estas construcciones, el estándar provee uno o más comportamientos posibles entre los cuales cada compilador puede elegir.

La situación es menos favorable para la mayor parte de estas construcciones del lenguaje, ya que el estándar no especifica el comportamiento ni hace requerimientos al compilador. El programa puede no compilar, terminar abruptamente, continuar funcionando con un comportamiento erróneo, e incluso terminar correctamente [3].

Al no imponer ningún requisito para los compiladores, normalmente se le da a estas construcciones, un comportamiento que permita implementar eficientemente aquellas otras cuyo comportamiento está completamente especificado, no advirtiendo al programador de su ocurrencia. Esto tiene como consecuencia la pérdida de portabilidad del programa, o peor aún, la posibilidad de errores al compilar el programa con diferentes compiladores por asumir un comportamiento consistente.

LLVM nace originalmente como una infraestructura de compilación diseñada para soportar un sistema de optimización *multi-etapa* [11]. Posteriormente, creció para incluir sub-proyectos como el compilador de *C*

Clang, entre otras herramientas [18, 13, 1, 12].

Es el objetivo de este trabajo la construcción de una herramienta que permita la detección de comportamientos indefinidos en tiempo de ejecución, utilizando como base *Clang* y *LLVM*.

2 MARCO TEÓRICO

En este capítulo se explicarán algunos conceptos esenciales para un buen entendimiento del trabajo.

EL LENGUAJE C

C es un lenguaje imperativo de propósito general con tipado estático. *C* provee constructores que mapean directamente a instrucciones típicas de *assembly*, lo que permitió hacer migraciones de aplicaciones antiguas como Sistemas Operativos u otras aplicaciones escritas para *sistemas empotrados*.

Fue desarrollado originalmente entre 1969 y 1973 por *Dennis Ritchie* en AT&T Bell Labs y fue posteriormente utilizado para re-implementar el sistema operativo *UNIX*. Desde entonces, *C* se ha vuelto uno de los lenguajes de programación más utilizados de todos los tiempos, con compiladores disponibles para casi cualquier arquitectura y sistema operativo. A partir de 1989 fue estandarizado por la *American National Standards Institute (ANSI)* y posteriormente también por la *International Organization for Standardization (ISO)*.

ESTÁNDAR ANSI C99

En 1983 el Instituto Nacional de Estándares Estadounidense *(ANSI)* formó un comité para establecer una especificación estándar del lenguaje *C*. El trabajo se completó en 1989, estableciendo así el primer estándar del lenguaje.

Después de que ANSI produjera aquel estándar oficial para el lenguaje *C* en 1989, se convirtió en estándar internacional en 1990. En 1995 se le hicieron algunas enmiendas que corrigieron algunos detalles de la versión de 1989 y se le agregó soporte para conjuntos de caracteres internacionales. Posteriormente, a fines de la misma década se le harían más revisiones que concluirían en la publicación de ISO/IEC 9899:1999 en 1999 [8], el cual sería adoptado por ANSI como estándar al año siguiente. El lenguaje definido por esa versión del estándar es normalmente referida como *"C99"*.

COMPORTAMIENTO INDEFINIDO

El estándar *C99* define *comportamiento* como *acción o apariencia externa* y define *comportamiento indefinido* de la siguiente forma [8, Sección 3, p. 3-4]:

> **undefined behavior**
>
> behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

Es decir, la acción o apariencia externa al usar un fragmento de programa erróneo o no portable, o al usar datos erróneos, para la cual el estándar no impone requisito alguno.

Además, el estándar agrega que los comportamientos indefinidos posibles van desde ignorar completamente la situación, con resultados impredecibles, hasta interrumpir la compilación o ejecución del programa con un mensaje de error, pasando por reaccionar de una manera documentada para el entorno, emitiendo o no una advertencia.

Esta completa ausencia de requisitos tiene como consecuencia que para un código fuente dado, diferentes compiladores tomen diferentes decisiones, produciendo programas resultantes diferentes. Dichas decisiones pueden estar guiadas por limitaciones de la plataforma, la decisión de mantener o no la compatibilidad con otro procesador o incluso la posibilidad de realizar optimizaciones durante la compilación o la ejecución del programa.

A lo largo del estándar se especifican las construcciones del lenguaje permitidas así como la semántica de las mismas. Algunas veces los comportamientos indefinidos son mencionados explícitamente, otras veces se dice que cualquier situación no definida es un comportamiento indefinido y en algunos casos se dan ejemplos. Sin embargo, el principal recurso para conocer los comportamientos indefinidos es la lista de definiciones por comprensión, con referencias a las secciones pertinentes, que el estándar provee en el *Anexo J.2* [8, p. 492]. Como ejemplo se da la siguiente definición:

> Conversion to or from an integer type produces a value outside the range that can be represented (6.3.1.4).

El comportamiento, al producir un valor fuera del rango representable mediante una conversión desde o a un tipo entero, es indefinido.

EL LENGUAJE C++

C++ es un lenguaje de programación de propósito general, basado en *C* con un énfasis en la programación de sistemas. Fue diseñado para soportar *abstracción de datos*, *programación orientada a objetos* y *programación genérica*, además de las técnicas soportadas por *C*, sin forzar un estilo particular de programación [21].

C++ fue desarrollado por Bjarne Stroustrup en Bell Labs desde 1979, basado en *C* por ser de propósito general, por su velocidad y portabilidad, incorporó características de *Simula* útiles para el desarrollo de sistemas grandes, así como otras influencias de *ALGOL 68*, *Ada*, *CLU* y *ML*. Fue inicialmente estandarizado por la *International Organization for Standardization (ISO)* en 1998, viendo sucesivas actualizaciones hasta su última versión publicada en 2014, conocida como *"C++14"*.

LLVM

LLVM es un conjunto de bibliotecas que proveen generación de *código de máquina* para múltiples plataformas de *hardware*, así como un optimizador de código independiente del lenguaje, facilitando la implementación de compiladores para nuevos lenguajes tanto como para lenguajes ya establecidos [18].

El proyecto *LLVM*, originalmente llamado *Low Level Virtual Machine*, fue iniciado en el año 2000 en la Universidad de Illinois bajo la dirección de Vikram Adve y Chris Lattner para investigar técnicas de compilación dinámicas ("Lifelong Program Analysis & Transformation") para programas arbitrarios [11, 13]. Escrito en *C++* y liberado bajo una licencia *permisiva*, pronto se convirtió en la base de muchos proyectos, tanto libres como privativos.

LLVM Intermediate Representation

En el centro de *LLVM* se encuentra la *LLVM Intermediate Representation* o *LLVM IR* que es la representación intermedia utilizada por todos los componentes del proyecto *LLVM*.

Es un lenguaje de programación de bajo nivel similar a *Assembly*, con un conjunto de instrucciones similares a las de un procesador, pero que abstrae los detalles de plataformas específicas. Tiene un sistema de tipos

sencillo pero que permite implementar la funcionalidad de lenguajes de alto nivel, así como un conjunto infinito de registros virtuales. Está disponible como una biblioteca de *C++* y cuenta con una representación textual legible por humanos [18, 11, 13].

CLANG

Clang es un compilador de *C*, *C++* y *Objective C*, que ofrece compilación veloz y bajo uso de memoria, mensajes de error claros y precisos y compatibilidad en la línea de comandos con *GCC* [16].

Está implementado como un conjunto de bibliotecas de *C++* que implementan todos los componentes del *front-end* de un compilador, como la división del código en sus componentes individuales, su análisis sintáctico y semántico, la construcción de estructuras de datos necesarias como la tabla de identificadores y el *AST*, etc; y utiliza *LLVM* como *back-end* para la generación de código de máquina [16, 1, 12].

ABSTRACT SYNTAX TREE

Un *Abstract Syntax Tree* o *AST* es una estructura de datos de tipo árbol, utilizada para representar la sintaxis de un programa, donde cada nodo interior corresponde a un operador u otras construcciones de un lenguaje y donde los hijos de cada nodo corresponden a los operandos del operador o a las partes semánticamente significativas de las construcciones [2].

Se dice que la sintaxis representada por el árbol es abstracta, ya que se prescinde de representar elementos de la sintaxis concreta, como, por ejemplo, indicaciones de precedencia o las palabras clave que conforman las construcciones del lenguaje.

Entonces, un *AST* para la expresión `(a + b) * c` tendría como raíz el nodo `*`, cuyos hijos serían los nodos `+` y `c`, y el nodo `+` tendía como hijos los nodos `a` y `b`. La precedencia queda reflejada en la estructura del árbol. Por otro lado, para una expresión como `if a then b else c`, un *AST* tendía como raíz el nodo `ifte`, cuyos hijos serían los nodos `a`, `b` y `c`, sin necesidad de incorporar las palabras claves de la expresión en el árbol.

COMPILADOR "SOURCE TO SOURCE"

Un compilador *source to source* toma como entrada código fuente escrito en un lenguaje de programación particular, y en lugar de producir código de máquina, produce como salida código fuente equivalente en un lenguaje de programación distinto.

Usualmente un compilador traduce código de un lenguaje de alto nivel a código en un lenguaje de más bajo nivel como es el caso del lenguaje de máquina. Sin embargo, un compilador *source to source* usualmente produce código fuente con un nivel de abstracción similar.

También puede ocurrir que el lenguaje de programación de salida sea el mismo que el de entrada, y que el compilador *source to source* se encargue, por ejemplo, de portar construcciones obsoletas de una versión del lenguaje a sus equivalentes de una versión más nueva, o se encargue de realizar en el código optimizaciones o correcciones.

INSTRUMENTACIÓN

Esta es una técnica que consiste en agregar a un programa o un sistema, código o instrucciones que permitan monitorear o medir distintas características durante su ejecución.

La *instrumentación* puede hacerse o en el código fuente del programa o en el programa ya compilado, y se puede utilizar para medir el desempeño del mismo, diagnosticar errores o rastrear el progreso de la ejecución. Esto permite, por ejemplo, ayudar a la optimización del programa identificando las partes del código más utilizadas, o verificar el estado del programa al ocurrir un error emitiendo información de diagnóstico por pantalla.

Las desventajas de esta técnica están en el costo agregado en tiempo de ejecución o el uso incrementado de recursos computacionales, debido al código agregado que debe ejecutarse, así como a las estructuras de datos necesarias para soportar la recolección de información. También es una limitación el hecho de que la información obtenida está restringida a las porciones del programa ejecutadas.

3 REALIZACIÓN

METODOLOGÍA DE TRABAJO

Concluida la revisión bibliográfica, el trabajo realizado se dividió en dos etapas. La primera fue la selección de comportamientos indefinidos y la creación de casos de prueba o *tests* para cada uno de ellos. La segunda fue la creación de un compilador y un *entorno de ejecución* que hiciera posible la detección de los comportamientos seleccionados.

Comportamientos indefinidos y casos de prueba

El *Anexo J.2* del estándar *C99* [8, p. 492] lista 191 comportamientos indefinidos. Intentar desde un principio la detección de tantos casos se juzgó imposible. Por lo tanto, el primer paso que se dio fue seleccionar un conjunto menor de comportamientos indefinidos.

El estándar *C99* define además del núcleo del lenguaje, es decir, cuáles son las construcciones permitidas por este y cuál es su semántica [8, Sección 6, p. 29], una biblioteca de funciones [8, Sección 7, p. 164]. De los 191 comportamientos listados, 98 corresponden a funciones de la biblioteca estándar. Estos fueron los primeros comportamientos indefinidos que se decidió descartar.

También se decidió descartar aquellos comportamientos indefinidos cuya ocurrencia estuviera determinada al momento de la compilación del programa, ya que la detección de estos casos es bastante común en los compiladores modernos. Esto deja aquellos cuya ocurrencia depende de algún elemento que sólo está presente durante la ejecución del programa, como valores ingresados por el usuario, o cuya detección requiere un análisis muy complejo del código fuente, como por ejemplo para determinar si un valor en un punto del programa produce un comportamiento indefinido al ser usado en otro punto del programa que no está directamente relacionado. Esta posible detección en tiempo de compilación es además muy propensa a los falsos positivos ya que la ocurrencia del comportamiento indefinido no depende completamente de la sintaxis del lenguaje.

Finalmente, se trató de eliminar todos los comportamientos indefinidos que dependieran muy estrechamente de detalles de la plataforma sobre la que se compila o se ejecuta un programa.

La lista de 93 comportamientos indefinidos correspondientes al núcleo de *C* quedó entonces reducida a una cantidad de, aproximadamente, 47 casos, cifra sujeta a las reinterpretaciones del significado de muchos casos que ocurrían con posteriores lecturas.

Esta clasificación demandó una lectura muy atenta de la definición de cada comportamiento indefinido, dada por comprensión, que se encuentra en el *Anexo J.2* del estándar *C99* [8, p. 492], así como también de la sección correspondiente del estándar. En esta se describe la semántica de las construcciones del lenguaje en las que ocurre el comportamiento indefinido en cuestión y por lo tanto los casos donde sí esta definido, es decir el conjunto complemento del dado por la definición del *Anexo J.2*.

Dicha lectura no resultó sencilla, ya que en muchos casos el texto se presta a múltiples interpretaciones, ya sea por el vocabulario técnico o la barrera idiomática; agravado esto por la falta de ejemplos concretos, casi total para los comportamientos indefinidos y notable en muchos casos para los comportamientos definidos.

Ya se había previsto que se requeriría alguna forma de poner a prueba la detección de los comportamientos indefinidos una vez se la implementara. Por lo tanto, se había decidido la creación de casos de prueba o *tests*: pequeños programas en *C*, escritos de modo que en cada uno ocurriera un comportamiento indefinido específico. De este modo se podría comprobar de manera controlada que la detección se realizaba exitosamente.

Dicha tarea de creación de *tests* se inició durante la clasificación de los comportamientos indefinidos y ayudó a la culminación de esta tarea. El ejercicio de implementar un programa que manifestara el comportamiento indefinido ayudaba a la comprensión del mismo y servía para verificar si era detectable en tiempo de compilación o no, al poder compilarlo con diferentes compiladores y verificar los resultados.

Al ser tan interdependiente con la comprensión de los comportamientos indefinidos, la creación de los *tests* fue muy iterativa. Para poder llevar cuenta del trabajo realizado y poder volver atrás cambios insatisfactorios se comenzó a utilizar un sistema de control de versiones. Este repositorio de *tests* fue utilizado en todo el desarrollo posterior de este trabajo para contener tanto el código producido, como la documentación escrita.

Compilador y entorno de ejecución

Una vez completa la clasificación de comportamientos indefinidos y la creación de casos de prueba se procedió con la implementación del compilador.

Esta implementación comenzó con la creación de pequeños experimentos en C++ utilizando la funcionalidad provista por las bibliotecas de *Clang*. Luego se determinó que implementar el compilador de esta manera sería muy dificultoso y no produciría los resultados deseados, por lo cual se procedió a la modificación de *Clang* mismo. Estos temas se abordarán en detalle en las siguientes secciones.

El primer paso para la realización de estas tareas consistió en obtener el código fuente de *LLVM* y *Clang* y compilarlo. El código fuente se obtuvo desde la *página web* del proyecto *LLVM*.

Otra decisión que se tomó fue la de mantener un entorno estable para la construcción de *Clang*. De este modo, cualquier cambio en el código compilado con el compilador modificado está garantizado de haber sido provocado por los cambios introducidos en el compilador y no por cambios introducidos en el sistema donde se lo construye. Para esta tarea se utilizó un sistema de *contenedores* de software con el fin de aislar el sistema de construcción de *Clang* y mantenerlo en las mismas versiones mientras durara el desarrollo.

Una vez completo este entorno de construcción para la versión modificada de *Clang*, se lo utilizó para construir una versión sin cambios. De este modo, se tiene disponible un medio de verificar que los resultados de compilar con la versión modificada no incluyan cambios indeseados.

Finalmente, para facilitar la detección de situaciones donde los cambios introducidos en *Clang* para detectar un comportamiento indefinido, causaran una regresión en otro comportamiento indefinido que ya era detectado, se creó un programa que, de manera automática, compila cada *test* con el compilador modificado, lo ejecuta y reporta si el comportamiento indefinido de cada uno fue detectado o no, como muestra la figura 3.1.

Utilizando todas estas herramientas se procedió a la implementación caso por caso de la detección de los comportamientos indefinidos, introduciendo modificaciones en *Clang* e implementando las distintas bibliotecas del entorno de ejecución.

```
[✓] 39.2_indUpInvVal_intAlign
[✓] 41.1_divBy0_int
[✓] 41.2_divBy0_uint
[✓] 41.3_divBy0_float
[✓] 41.4_modBy0_int
[✓] 41.5_modBy0_uint
[✓] 42.1_addBeyondArray
[✓] 42.2.1_addBeyondArray
[✓] 42.2.2_addBeyondArray
[✓] 42.3_addBeyondArray
[✓] 42.4_addBeyondArray
[✓] 42.5_addBeyondArray
[✓] 42.6_addBeyondArray
[✓] 42.7_addBeyondArray
[✓] 43_addJustBeyondArray
[✓] 44_pointerSubsDiffArray
[✓] 45.1_subscriptOutOfRange
[✓] 45.2_subscriptOutOfRange
[✓] 45.3_subscriptOutOfRange
[✓] 45.4_subscriptOutOfRange
[✓] 45.5_subscriptOutOfRange
[✓] 45.6_subscriptOutOfRange
[✓] 45.7_subscriptOutOfRange
[✓] 45.8_subscriptOutOfRange
[✓] 46.1_ptrDiff
[✓] 46.2_ptrDiff
[✓] 47.1_expShftNeg-Over
[✓] 47.2_expShftNeg-Over
[✓] 47.3_expShftNeg-Over
[✓] 47.4_expShftNeg-Over
[✓] 48.1_sgndPrmtdShft
[✓] 48.2_sgndPrmtdShft
[✓] 49.1_prtRelComp
[✓] 49.2_prtRelComp
[✓] 49.3_prtRelComp
[✗] 50.1_assignOverlap
[✗] 50.2_assignOverlap
[!] 58_flexArray
[✗] 60_modConstWPntr
[✗] 61_accVolatWPntr
[✗] 64_accessRestrictPointer
[✓] 68_arraySizeExpNotConst
[✗] 69_arrayTypesIncompElemUneqSize
[✗] 70_arrayParStatic
[✗] 72_incompatFunctionTypes
[✗] 73_unnamedMemberValueUsed
[†] 75_invalidStructInitializer:
 └➤ clang -std=c99 ./tests/75_invalidStructInitializer.c -o /tmp/a.out -Lruntime
/lib -lchkdecl-stub -lchkopsbtwszeroneg-stub -lchkopsderef-stub -lchkopsdmzero-s
tub -lchkopsptraddsub-stub -lchkopsptrcmp-stub -lchkopsptrdiff-stub -lchkopssgnd
shft-stub -lchkopsshfnegover-stub -lchksubscript-stub -lfixopsptrasgn-stub -lfix
opsptrcmp-stub -lregdecl-stub
[✗] 81_{isReachedAndValueUsedbyCaller
57 PASSED, 26 FAILED, 4 Unknown | 0.66:1
```

Figura 3.1: *Test suite* en ejecución

PRIMER ENFOQUE: *libtooling* Y *libastmatchers*

Una vez completa la selección de comportamientos indefinidos y habiendo creado un conjunto de pequeños programas que evidenciaban dichos comportamientos, se dio comienzo a la etapa de implementación

de este trabajo. Esta consistió en la creación de un compilador de *C* que hiciera posible la detección de comportamientos indefinidos.

Para llevar a cabo dicha implementación se había planeado desde el comienzo utilizar el compilador *Clang* como base. Este compilador está diseñado como un conjunto de bibliotecas que proveen acceso a su funcionalidad, como el *parseo* del código, la generación del *AST* correspondiente y su transformación en código *LLVM IR*, y a la funcionalidad provista por *LLVM*, como la generación de *código de máquina* y *linkeo* del programa [16, 1, 12]. Esta característica lo hace un excelente recurso cuando se trata de implementar herramientas para analizar o manipular código *C*.

Por lo tanto, el primer enfoque de la etapa de implementación consistió en la utilización de las bibliotecas de *C++* que proveen la funcionalidad de *Clang*. Estas permitirían crear una herramienta sencilla y concisa, con la que instrumentar el código de un programa en *C* para hacer posible la detección de comportamientos indefinidos, sin la necesidad de descender al complejo código fuente de un compilador.

La primera biblioteca que fue de interés es *libtooling*, que permite crear fácilmente herramientas independientes de *Clang*. La segunda es *libastmatchers*, que provee una forma concisa y sencilla de describir patrones de la *AST* de modo que se puede encontrar fácilmente todo fragmento de código que tenga las características requeridas. Otras bibliotecas de *Clang* también fueron usadas durante este primer enfoque para complementar a las recién descritas.

Utilizando las bibliotecas mencionadas se creó una serie de experimentos basados en ejemplos disponibles en la documentación de *Clang* [17]. Para estos experimentos se eligió el más sencillo de los comportamientos indefinidos a detectar, la división y el módulo con un denominador que vale 0:

> The value of the second operand of the / or % operator is zero.

Los primeros de estos experimentos imprimían recursivamente una representación textual de todos los nodos del *AST* utilizando *libastmatchers* y *libtooling*. Luego de unas pocas iteraciones se llegó a un programa que sólo imprimía recursivamente los nodos correspondientes a los operadores de la división y el módulo:

```
StatementMatcher DivModMatcher =
↪    binaryOperator(anyOf(hasOperatorName("/"),
↪    hasOperatorName("%"))).bind("op");
```

Lo primero que se hace es definir un objeto de la clase de C++ de *Clang* `StatementMatcher` que permite encontrar y *ligar* nodos específicos. Esto se hace utilizando un lenguaje de dominio específico (*domain specific language* o *DSL*) provisto por *libastmatchers* [17].

```
class DivModPrinter : public MatchFinder::MatchCallback {
public :
  virtual void run(const MatchFinder::MatchResult &Result) {
    const BinaryOperator *Op =
↪     Result.Nodes.getNodeAs<clang::BinaryOperator>("op");
    if (Op && Result.SourceManager->isFromMainFile(Op->getLocStart())) {
      Op->dumpColor();
    }
  }
};
```

A continuación se crea una subclase de `MatchCallback`. Esta clase tiene un método `run` que es llamado cuando un `StatementMatcher` encuentra un nodo del *AST*. A este método se le pasa una referencia a una estructura de tipo `MatchResult`. Mediante esta estructura primero se obtiene un puntero al nodo encontrado, luego se controla que el operador encontrado pertenezca al archivo principal del programa para minimizar la cantidad de resultados y finalmente se imprime la representación textual del nodo.

```
int main(int argc, const char **argv) {
  CommonOptionsParser OptionsParser(argc, argv);
  ClangTool Tool(OptionsParser.getCompilations(),
↪     OptionsParser.getSourcePathList());

  DivModPrinter Printer;
  MatchFinder Finder;
  Finder.addMatcher(DivModMatcher, &Printer);

  return Tool.run(newFrontendActionFactory(&Finder));
}
```

Para completar el experimento se instancian un objeto de la clase `ClangTool` y un objeto de la clase `MatchFinder`; se agregan a este el objeto de la clase `StatementMatcher` declarado al principio junto a un objeto de la

subclase de `MatchCallback` creada. Finalmente, mediante el método `newFrontendActionFactory` se completa el trabajo necesario para que el objeto `ClangTool` genere el *AST* y utilice el objeto `MatchFinder` para encontrar e imprimir los nodos deseados.

```
#include <stdio.h>

int main(void) {
    int a = 5;
    int b = 6;
    printf("%d\n", a / b);
    printf("%d\n", a % b);
    return 0;
}
```

El resultado de procesar el código *C* anterior con este primer experimento es el siguiente:

```
BinaryOperator 0x29d5998 'int' '/'
|-ImplicitCastExpr 0x29d5968 'int' <LValueToRValue>
| `-DeclRefExpr 0x29d48d8 'int' lvalue Var 0x29d4750 'a' 'int'
`-ImplicitCastExpr 0x29d5980 'int' <LValueToRValue>
  `-DeclRefExpr 0x29d5940 'int' lvalue Var 0x29d47f0 'b' 'int'
BinaryOperator 0x29d5b40 'int' '%'
|-ImplicitCastExpr 0x29d5b10 'int' <LValueToRValue>
| `-DeclRefExpr 0x29d5ac0 'int' lvalue Var 0x29d4750 'a' 'int'
`-ImplicitCastExpr 0x29d5b28 'int' <LValueToRValue>
  `-DeclRefExpr 0x29d5ae8 'int' lvalue Var 0x29d47f0 'b' 'int'
```

Para ilustrar mejor la naturaleza recursiva del recorrido del *AST* se pueden reemplazar en el código *C* ambas llamadas a la función `printf` por una única llamada como la siguiente `printf("%d\n", (a / b) % 2);`. La salida producida en este caso es la siguiente:

```
BinaryOperator 0x2a8da20 'int' '%'
|-ParenExpr 0x2a8d9e0 'int'
| `-BinaryOperator 0x2a8d9b8 'int' '/'
|   |-ImplicitCastExpr 0x2a8d988 'int' <LValueToRValue>
|   | `-DeclRefExpr 0x2a8c8f8 'int' lvalue Var 0x2a8c770 'a' 'int'
|   `-ImplicitCastExpr 0x2a8d9a0 'int' <LValueToRValue>
```

```
|     '-DeclRefExpr 0x2a8d960 'int' lvalue Var 0x2a8c810 'b' 'int'
'-IntegerLiteral 0x2a8da00 'int' 2
BinaryOperator 0x2a8d9b8 'int' '/'
|-ImplicitCastExpr 0x2a8d988 'int' <LValueToRValue>
| '-DeclRefExpr 0x2a8c8f8 'int' lvalue Var 0x2a8c770 'a' 'int'
'-ImplicitCastExpr 0x2a8d9a0 'int' <LValueToRValue>
  '-DeclRefExpr 0x2a8d960 'int' lvalue Var 0x2a8c810 'b' 'int'
```

Se puede ver cómo la representación textual del nodo correspondiente a la división aparece primero como uno de los operandos del nodo correspondiente al módulo y luego por sí sola.

La siguiente iteración de los experimentos buscaba obtener el fragmento de código correspondiente a los nodos encontrados. Para esto se modificó la subclase de `MatchCallback` creada de la siguiente forma:

```
class DivModPrinter : public MatchFinder::MatchCallback {
public :
  virtual void run(const MatchFinder::MatchResult &Result) {
    const BinaryOperator *Op =
↪     Result.Nodes.getNodeAs<clang::BinaryOperator>("op");
    Rewriter Rw(*Result.SourceManager, Result.Context->getLangOpts());
    if (Op && Result.SourceManager->isFromMainFile(Op->getLocStart())) {
      std::cout << Rw.ConvertToString((BinaryOperator *) Op) <<
↪     std::endl;
    }
  }
};
```

Se agregó al método `run` una instancia de la clase `Rewriter`. Esta permite obtener de un objeto del *AST*, el código *C* correspondiente, así como introducir modificaciones. La salida que produce este experimento para el código *C* usado como ejemplo para la versión anterior es el siguiente:

```
a / b
a % b
```

Y para la versión con un único `printf`:

```
(a / b) % 2
a / b
```

Habiendo llegado a este punto se decidió intentar crear un compilador *source to source* que instrumentara el código para detectar el comportamiento indefinido de la división y el módulo. Esto presentó dos dificultades.

La primera dificultad encontrada fue lograr reproducir el código fuente completo sin repeticiones ni omisiones. Debido al recorrido recursivo del *AST* y que cada llamado al método `run` del objeto `MatchCallback` es independiente, es imposible crear un objeto `StatementMatcher` genérico y evitar que se reproduzca el código correspondiente de un nodo y luego el correspondiente a sus nodos hijos. También es difícil crear un conjunto de objetos `StatementMatcher` específicos que cubran todas las posibles líneas de un programa.

La solución a este problema consistió en sacar el objeto `Rewriter` del método `run` y convertirlo en un parámetro de la instanciación del objeto `MatchFinder`. De esta forma sólo es necesario localizar los nodos correspondientes a la división y el módulo y realizar los cambios necesarios. Cuando la búsqueda está completa se obtiene el código *C* modificado completo mediante el objeto `Rewriter`. La nueva definición de la subclase de `MatchCallback` se ve a continuación:

```
class DivModInstrumenter : public MatchFinder::MatchCallback {
public :
  explicit DivModInstrumenter(Rewriter &Rwtr) : Rw(Rwtr) {}

  virtual void run(const MatchFinder::MatchResult &Result) {
    const BinaryOperator *Op =
↪    Result.Nodes.getNodeAs<clang::BinaryOperator>("op");
    if (Op && Result.SourceManager->isFromMainFile(Op->getLocStart())) {
      const Stmt *St = Result.Context->getParents(*Op)[0].get<Stmt>();
      const Stmt *CS = Result.Context->getParents(*St)[0].get<Stmt>();
      while (strcmp(CS->getStmtClassName(), "CompoundStmt") != 0) {
        St = CS;
        CS = Result.Context->getParents(*CS)[0].get<Stmt>();
      }
      std::string check = "assert(" + Rw.ConvertToString((Stmt *)
↪     Op->getRHS()) + "!=0);\n";
      Rw.InsertText(St->getLocStart(), check, true, true);
    }
  }
```

```
private :
  Rewriter &Rw;
};
```

Como se mencionó, la clase ahora recibe un argumento de tipo `Rewriter`. Además, en el método `run` se itera sobre los padres del operador encontrado hasta que se encuentre uno perteneciente a la clase `CompoundStmt`, que corresponde a un bloque de código C `{ ... }`. Esto asegura que no se introduce código a mitad de la expresión donde ocurre el operador encontrado. Se conserva un puntero al nodo encontrado justo antes del `CompoundStmt` para poder hacer los cambios dentro de este último. Luego se genera en un *string* una llamada a la función `assert` que compara el lado derecho de la operación (*right hand side* o *RHS*) con 0. Finalmente se inserta este *string* antes de la expresión que contiene la operación de división o módulo.

Lamentablemente, para poder instanciar un objeto de la clase `Rewriter` fuera del método `run` fue necesario renunciar al objeto `ClangTool` e instanciar todos los objetos necesarios para replicar su funcionalidad. La función `main` resultante es considerablemente más compleja:

```
int main(int argc, const char **argv) {
  CompilerInstance CI;
  CI.createDiagnostics(0, false);
  CI.getTargetOpts().Triple = llvm::sys::getDefaultTargetTriple();
  CI.setTarget(TargetInfo::CreateTargetInfo(CI.getDiagnostics(),
↪    &CI.getTargetOpts()));
  CI.createFileManager();
  CI.createSourceManager(CI.getFileManager());
  CI.createPreprocessor();
  CI.createASTContext();
  SourceManager &SM = CI.getSourceManager();
  ...
```

La mayor parte de los objetos necesarios se crean a través de un objeto de la clase `CompilerInstance`. Con el código anterior se tiene lo necesario para instanciar el objeto `Rewriter` además del objeto `MatchFinder` que se instancia de manera análoga a la usada en versión anterior:

```
  ...
  Rewriter Rw(CI.getSourceManager(), CI.getLangOpts());
```

```
  DivModInstrumenter Instrumenter(Rw);
  MatchFinder Finder;
  Finder.addMatcher(DivModMatcher, &Instrumenter);
  ...
```

A continuación se ve el código que reemplaza la invocación del método `run` del objeto `ClangTool` y el método `newFrontendActionFactory`:

```
  ...
  SM.createMainFileID(CI.getFileManager().getFile(argv[1]));
  CI.getDiagnosticClient().BeginSourceFile(CI.getLangOpts(),
↪    &CI.getPreprocessor());
  ParseAST(CI.getPreprocessor(), Finder.newASTConsumer(),
↪    CI.getASTContext());
  CI.getDiagnosticClient().EndSourceFile();
  ...
```

Finalmente, se obtiene del objeto `Rewriter` el código C modificado, se agrega `#include <assert.h>` al comienzo y se lo imprime.

```
  ...
  Rw.InsertTextBefore(SM.getLocForStartOfFile(SM.getMainFileID()),
↪    "#include <assert.h>\n");
  const RewriteBuffer *Buff =
↪    Rw.getRewriteBufferFor(SM.getMainFileID());
  llvm::outs() << std::string(Buff->begin(), Buff->end());

  return 0;
}
```

Como se puede ver, esta versión es considerablemente más compleja que la previa, y a pesar de esto no puede replicar completamente la funcionalidad anterior. Para empezar, esta versión está limitada a procesar un único archivo con código C por invocación, mientras que la anterior no lo estaba. Además, no se ha configurado el preprocesador y por lo tanto emite un error de *archivo no encontrado* por cada `#include` del código.

A continuación está el código resultante de procesar el ejemplo usado anteriormente:

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

int main(void) {
    int a = 5;
    int b = 6;
    assert(b!=0);
    printf("%d\n", a / b);
    assert(b!=0);
    printf("%d\n", a % b);

    return 0;
}
```

La salida del prototipo correspondiente a la versión con un único `printf` tiene las siguientes diferencias:

```
...
assert(2!=0);
assert(b!=0);
printf("%d\n", (a / b) % 2);
...
```

Parece funcionar correctamente, sin embargo si cambiamos la asociatividad de la expresión donde ocurren los operadores los resultados no son tan buenos:

```
...
assert((b % 2)!=0);
assert(2!=0);
printf("%d\n", a / (b % 2));
...
```

Si en este caso el lado derecho de la operación módulo no fuera una constante distinta de 0, la detección del comportamiento podría fallar al ejecutar el primer `assert`. Claramente la herramienta construida sólo funciona correctamente con el código *C* más sencillo.

Esta fue la última versión producida y luego de evaluar los resultados se decidió abandonar este enfoque. Sin embargo, el motivo no fue el pobre desempeño de este prototipo.

El programa en *C* emitido por este prototipo era un programa diferente del original, tanto sintácticamente como semánticamente. El siguiente ejemplo ilustra el problema:

```
#include <stdio.h>

int f(void) {
    /* unknown */
}

int main(void) {
    int a = 5;
    printf("%d\n", a / f());

    return 0;
}
```

En este programa el denominador de la división es el valor devuelto por una función cuyo funcionamiento se desconoce. En el código devuelto por el prototipo se puede ver lo siguiente:

```
assert(f()!=0);
printf("%d\n", a / f());
```

La llamada a la función `f` se encuentra duplicada. La función se llamará dos veces, posiblemente devolviendo valores distintos en cada ocasión y cambiando el estado del programa de maneras desconocidas. Para poder detectar con precisión si un comportamiento indefinido ocurre o no es necesario comprobar si el mismo valor que luego será usado como denominador es 0 o no. Esto es posible en código de más bajo nivel, como el correspondiente a la división del ejemplo anterior que se ve a continuación:

```
%call = call i32 @f()
%div = sdiv i32 %0, %call
```

Este es el código utilizado como representación intermedia por *Clang*, *LLVM IR*. Se decidió entonces que la próxima etapa de este trabajo sea la instrumentación del código *LLVM IR* producido por *Clang*.

ENFOQUE DEFINITIVO: *codegen*

Habiendo comprobado la dificultad de crear un compilador, incluso contando con la funcionalidad necesaria en bibliotecas preexistentes, se decidió que era más viable la modificación de un compilador existente. Queriendo también acceso a instrucciones de bajo nivel se decidió modificar la etapa de generación de código intermedio del mismo.

El trabajo realizado por un compilador se realiza en varias etapas, y entre cada una de ellas pueden generarse distintas representaciones intermedias del programa compilado [2]. En el caso de *Clang*, antes de la generación de código de máquina realizada por *LLVM*, se crea a partir del *AST*, una representación intermedia del programa en el lenguaje *LLVM IR*.

Clang está organizado como un conjunto de bibliotecas y el código fuente de este está organizado en carpetas, cada una correspondiente a una biblioteca. La biblioteca encargada de la generación de código *LLVM IR* se llama *libcodegen* y la carpeta que le corresponde se llama *CodeGen*.

Dentro de *CodeGen* existen varios módulos, cada uno encargado de la generación del *LLVM IR* de una parte de *C* y los otros lenguajes soportados por *Clang*.

Los módulos principales son *CodeGenModule*, *CodeGenTypes* y *CodeGenFunction*, que definen clases de *C++* homónimas y sirven como punto de entrada para emitir el código *LLVM IR* de módulos, tipos y funciones, respectivamente. También definen muchos métodos auxiliares utilizados por el resto de los módulos de *CodeGen* y proveen acceso a clases y métodos de *LLVM* para emitir y manipular instrucciones y objetos de *LLVM IR*.

El resto de los módulos de *CodeGen* implementan la mayor parte de la funcionalidad necesaria para la generación de código *LLVM IR*. De particular interés para este trabajo son el módulo encargado de emitir el código intermedio correspondiente a declaraciones de funciones y variables, llamado *CGDecl*, y el módulo encargado de emitir las expresiones, llamado *CGExpr*, o más específicamente el submódulo *CGExprScalar*. En este módulo es donde se emite el código intermedio correspondiente a operadores aritméticos, lógicos y de acceso a objetos cuya semántica es un valor entero o de punto flotante, incluidos punteros y direcciones de memoria. Toda la intervención realizada en *CodeGen* se hizo en estos dos módulos, que tiene como punto de entrada el módulo *CodeGenFunction*.

Para minimizar el número de líneas insertadas o modificadas en los módulos anteriores y mantener la mayor parte de los cambios aislados, se creó un nuevo módulo dentro de *CodeGen*. En este se definió una

clase que extiende la clase `CodeGenFunction` con los métodos definidos para emitir el código *LLVM IR* necesario para la detección de comportamientos indefinidos. De esta forma, en el lugar donde se desea incluir dicho código sólo es necesario incluir una declaración como la siguiente:

```
IDefCGF &ICGF = static_cast<IDefCGF &>(*this);
```

Aquí se *castea* el objeto `CodeGenFunction` apuntado por `this` en un objeto de la clase nuevamente declarada. De esta forma, en los métodos con que se extiende el objeto `CodeGenFunction` se tiene completo acceso a los métodos auxiliares y de generación de *LLVM IR* que este provee. A continuación, sólo es necesario llamar al método definido para generar el código de verificación:

```
ICGF.emitChkArraySizeNegCall(elementCount, D.getLocation());
```

En las siguientes secciones se detallará el proceso de la creación de estas funciones. Sin embargo, se puede mencionar brevemente como se llegó al esquema general que siguen.

Los primeros intentos exitosos de manipular el código *LLVM IR* generado consistieron en reemplazar el resultado de una operación aritmética por un valor constante. Simplemente se reemplazó la instrucción que emitía la operación, en este caso `CreateSDiv`, por una instrucción que emitía una constante como `getInt32`. Sin embargo, a la hora de generar construcciones más complejas como condicionales, necesarios para verificar si el denominador de la división vale 0, *LLVM IR* probó ser mucho más complejo.

Para evitar esta complejidad se decidió que el código que debía determinar si un comportamiento indefinido ocurre o no, fuera escrito como funciones en C, compiladas de manera independiente, y en *CodeGen* generar llamados a dichas funciones que luego serían *linkeadas*. Con este objetivo pronto se encontró el método `CreateCall` entre los métodos de *LLVM* a los que el objeto `CodeGenFunction` provee acceso.

Teniendo un método viable para instrumentar un programa se decidió proceder con la implementación de la detección de tantos comportamientos indefinidos como fuera posible, trabajo sobre el que se profundiza en las secciones siguientes. De ahora en más, en este trabajo se referirá al compilador modificado *Clang* como `idbcc` y al conjunto de módulos verificadores como `idbre`. A la unión de estos se la llamará *IDefBehav*.

Primeros casos, `div` y `rem`

Los primeros 2 casos a resolver fueron, por su simpleza y semejanza, los casos de `div` y `rem` operados con 0 como divisor (en su segundo argumento). En el caso de la división eso es un comportamiento indefinido porque la división matemática no se encuentra definida para el caso en que el divisor sea el entero 0. Los compiladores usuales no realizan ningún tipo de chequeo respecto a este valor y lo pasan derecho al procesador. Si el procesador implementa alguna especie de *excepción* para ese caso, entonces el error es devuelto y el programa se detiene; pero si el procesador no hiciera ningún tipo de verificación el resultado esperado puede ser realmente cualquier cosa; no habiendo ninguna garantía de que todos los procesadores de todas las arquitecturas realicen tales verificaciones.

Para este caso, el compilador `idbcc` sólo necesitó agregar un llamado a función justo antes de realizar la operación de división. Dicho llamado recibe como único argumento el segundo argumento de la división y controla que el mismo sea distinto de 0.

A modo de ejemplo, el código necesario para realizar tal chequeo fue:

```
Value *FilePos = emitFileLoc(Loc);
...
llvm::Type *ArgTypes[] = { OpRHS->getType(), Int8PtrTy };
llvm::FunctionType *ChkZeroTy =
  llvm::FunctionType::get(VoidTy, ArgTypes, false);
// Fun is either "chk_fzero", "chk_uzero" and "chk_szero",
// for float, unsigned and signed values respectively.
Value *ChkZero = CGM.CreateRuntimeFunction(ChkZeroTy, Fun);

Builder.CreateCall2(ChkZero, OpRHS, FilePos);
```

Este fragmento de código simplemente se encarga de tomar la expresión del lado derecho del operador `div` y su posición en el código fuente original.

De allí obtiene el tipo de dicha expresión y con ella construye la tupla de tipos que la función chequeadora del entorno `idbre` recibirá, agregándole el tipo de otros argumentos que hagan falta pasar, en este caso, el *string* con la posición en el código fuente original para referencia en el reporte del error. Así se guarda en `ArgTypes` o bien la tupla `(float, char *)`, o `(double, char *)`, o `(int, char *)`, o `(unsigned int, char *)`, etc., según

el tipo de la expresión del lado derecho del operador `div`. Una vez construida la tupla de argumentos, se construye el tipo completo del llamado a función que estamos creando, esto es, la tupla `ArgTypes` junto con el valor devuelto por la función del entorno `idbre`, que en este caso es `void`, y todo eso es almacenado en `ChkZeroTy`. Luego, se genera el objeto de función con todos los datos generados, el tipo completo que se acaba de construir con su valor de devolución, y el nombre de la función en el *runtime*. Por último, se crea el llamado uniendo el objeto de función recién generado con los argumentos que interesan pasar, la expresión derecha y su posición en el código fuente.

El fragmento de código recién mostrado se encuentra invocado por generador de *LLVM IR* de Clang cuando éste se encuentra con la operación división. La función en Clang entonces invocada, y que ahora se encargará además de realizar lo recién expuesto en código, es la función `Value *ScalarExprEmitter::EmitDiv(const BinOpInfo &Ops)` y luce de la siguiente manera:

```
Value *ScalarExprEmitter::EmitDiv(const BinOpInfo &Ops) {
  if ((CGF.SanOpts->IntegerDivideByZero ||
       CGF.SanOpts->SignedIntegerOverflow) &&
      Ops.Ty->isIntegerType()) {
    llvm::Value *Zero =
↪     llvm::Constant::getNullValue(ConvertType(Ops.Ty));
    EmitUndefinedBehaviorIntegerDivAndRemCheck(Ops, Zero, true);
  } else if (CGF.SanOpts->FloatDivideByZero &&
             Ops.Ty->isRealFloatingType()) {
    llvm::Value *Zero =
↪     llvm::Constant::getNullValue(ConvertType(Ops.Ty));
    EmitBinOpCheck(Builder.CreateFCmpUNE(Ops.RHS, Zero), Ops);
  }

  IDefCGF &ICGF = static_cast<IDefCGF &>(CGF);

  if (Ops.LHS->getType()->isFPOrFPVectorTy()) {
    ICGF.emitChkZeroCall("chk_fzero", Ops.RHS, Ops.E->getExprLoc());
    llvm::Value *Val = Builder.CreateFDiv(Ops.LHS, Ops.RHS, "div");
    if (CGF.getLangOpts().OpenCL) {
      // OpenCL 1.1 7.4: minimum accuracy of single precision / is 2.5ulp
      llvm::Type *ValTy = Val->getType();
```

```
      if (ValTy->isFloatTy() ||
          (isa<llvm::VectorType>(ValTy) &&

↪    cast<llvm::VectorType>(ValTy)->getElementType()->isFloatTy()))
        CGF.SetFPAccuracy(Val, 2.5);
    }
    return Val;
  }
  else if (Ops.Ty->hasUnsignedIntegerRepresentation()) {
    ICGF.emitChkZeroCall("chk_uzero", Ops.RHS, Ops.E->getExprLoc());
    return Builder.CreateUDiv(Ops.LHS, Ops.RHS, "div");
  } else {
    ICGF.emitChkZeroCall("chk_szero", Ops.RHS, Ops.E->getExprLoc());
    return Builder.CreateSDiv(Ops.LHS, Ops.RHS, "div");
  }
}
```

Se puede ver que este método de *C++* tiene varios puntos de salida (según el tipado de la división), y en cada uno de ellos es precedido por la inserción del llamado al chequeador en el entorno `idbre`.

Con estas modificaciones en el generador de código *LLVM IR*, al usar `idbcc` para compilar un programa que contenga una división, se puede observar cómo en la representación intermedia ya se genera el llamado deseado:

```
%2 = load i32* %a, align 4
call void @chk_szero(i32 %2, i8* getelementptr inbounds ([12 x i8]* @1,
↪    i32 0, i32 0))
%div = sdiv i32 1, %2
```

habiéndose agregado la línea completa que realiza el `call void @chk_szero(...)`, que antes de la modificación del compilador no se generaba.

La línea de compilación utilizada para visualizar este comportamiento con la modificación del código es:

```
$ idbcc -std=c99 41.1_divBy0_int.c -S -emit-llvm -o 41.1_divBy0_int.ll
```

siendo `idbcc` en este llamado el compilador Clang modificado de *IDefBehav*. No es necesario siempre generar el archivo de salida de *LLVM IR* pero es una herramienta útil para encontrar si la inserción de los llamados al entorno `idbre` están siendo efectuados correctamente.

Por último, la implementación de la función en el entorno `idbre` que es *linkeada* contra el programa a verificar es el siguiente fragmento de código escrito en *C*:

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

void chk_szero(int a, const char *fpos) {
    // The value of the second operand of the / or % operator is zero
↪    (6.5.5).

    if (a == 0) {
        fprintf(stderr, "%s: Second operand of a / or %% is zero.\n",
↪    fpos);
        abort();
    }
}
```

El caso del cómputo del resto (módulo) es idéntico a la división salvo porque no necesita ser realizado cuando las expresiones son de tipo flotante ya que el operador `rem` sólo se encuentra definido para valores enteros. Como se vio en el código de `idbcc` encargado de generar el llamado al chequeador, el método en cuestión tenía 3 posibilidades dependiendo de si la división era una división entera con signo, entera sin signo o de punto flotante. El cálculo del módulo sólo tiene 2.

Una preocupación que puede surgir, es si vuelve a ocurrir lo visto en la sección 3.2, de que se dupliquen llamados, provocando una doble evaluación de la expresión en el divisor. Afortunadamente esto no ocurre debido a que cuando se genera el código *LLVM IR* ya todas las expresiones han sido pre-evaluadas para ser representadas en forma *SSA* (static single assignment form). Es decir, que al punto de generar la división, tanto dividendo como divisor son insertadas en la *IR* como expresiones ya evaluadas. Lo que recibe el chequeador entonces es una copia exacta del mismo registro *SSA* que recibirá después la división, en este caso, `%2`.

Los operadores Shift *derecho e izquierdo*

Los operadores de *Shift* derecho (`>>`) e izquierdo (`<<`) en el lenguaje *C* son operadores que se utilizan para desplazar los bits de una expresión hacia izquierda o derecha (según el operador utilizado) una cantidad dada;

así si se tiene `a<<5` se estará desplazando todos los bits de la expresión en `a` la cantidad de `5` lugares hacia la izquierda. El valor con el que es rellenada la expresión depende de si el desplazamiento es hacia derecha o hacia izquierda, y del signo de la expresión `a`.

Mencionado el funcionamiento de los operadores de *Shift*, los comportamientos indefinidos que afectan a dichos operadores son más de uno:

> An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression.

que dice que es un comportamiento indefinido si se utiliza el operador de *Shift* para desplazar una expresión en una cantidad *negativa* de posiciones o una cantidad mayor al ancho en bits del tipo de la expresión.

Y un segundo comportamiento:

> An expression having signed promoted type is left-shifted and either the value of the expression is negative or the result of shifting would be not be representable in the promoted type.

que dice que si la expresión alcanzada por el operador de *Shift* izquierdo tiene *tipo* con signo, y además ocurre alguna de las siguientes: la expresión es *negativa* o el resultado del desplazamiento no puede ser representado en el tipo promovido, entonces el resultado de la operación es indefinido también. Cabe aclarar que en este caso, como el tipo de la expresión es con signo, se puede utilizar un bit menos que el ancho de bits del tipo en el desplazamiento, dado que ese bit es el que se utiliza para la representación del signo.

La especificación completa del uso de ambos operadores se puede encontrar en la sección *Bitwise shift operators* del estándar [8, (6.5.7)].

Para el primero de los comportamientos el tratamiento es sencillo, al momento de generar la instrucción *LLVM IR* se computa el tamaño en bits de la expresión. A esta altura, *LLVM* todavía es capaz de proveer gran parte de la información del objeto. Para eso se llama al método del objeto en *LLVM* `LHS->getType()->getIntegerBitWidth()` que contiene el dato, siendo `LHS` el objeto que representa la expresión que va a ser *Shifteada*. Con el dato de la longitud en bits del tipo y el otro argumento del operador *Shift*, `RHS` (la cantidad de bits a desplazar), es cuestión de realizar dos chequeos: primero, que el valor en `RHS` sea mayor o igual que `0` y segundo, que sea menor estricto que la cantidad de bits que se necesitan para representar

a `LHS`; si alguna de esas dos condiciones no se cumpliera, el entorno de ejecución aborta con el mensaje de error correspondiente.

Como bien se vio en la explicación de cada comportamiento, el segundo de ellos sólo hay que verificarlo cuando la expresión a la que se le aplica la operación de *Shifting* (`LHS`) es de tipo con signo, es decir que pueda tomar valores negativos, aunque el valor actual no lo sea. Entonces, sólo se realizan los chequeos si se puede verificar que `Ops.Ty->hasSignedIntegerRepresentation()` es cierto, siendo `Ops` el objeto que representa al operador binario y `.Ty` el campo con su tipo.

En caso de ser cierto se guardan, similarmente al caso anterior, los objetos que representan la cantidad de bits a *Shiftear*, el tamaño de los bits necesarios para representar el *tipo* de la expresión afectada por la operación de *Shift* (`RHS` y bits del `LHS` respectivamente); pero además se conserva el objeto mismo que representa la expresión a ser *Shifteada* `LHS`.

Una vez obtenidos todos estos datos se hicieron dos chequeos: el primero y más simple, verificar que la expresión no sea menor que `0`; el segundo chequeo requirió un poco más de estrategia. Este chequeo debe controlar que el resultado de *Shiftear* la expresión sea representable en el *tipo* de la expresión, y para ello se utilizó el siguiente fragmento de código en el verificador del *comportamiento*:

```
while (expr != 0) {
    expr = expr >> 1;
    count++;
}
```

Como se puede observar en esta sección del verificador se obtienen la cantidad de bits necesarios para representar el valor de la expresión `expr`, que es la expresión siendo sometida a la operación de *Shift*. Una vez que se obtiene la cantidad de bits utilizados, se verifica que esa cantidad, añadida al valor de *Shift* con el se está queriendo operar `expr`, no resulte mayor o igual que la cantidad de bits de la que se dispone para representar el *tipo* de la expresión y en caso de no cumplir con tal requerimiento, una vez más el verificador aborta. Si resulta en el valor de igualdad también se incurre en el comportamiento indefinido porque, como se mencionó con anterioridad, se dispone de un bit menos para representar los valores del *tipo* de la expresión siendo ese bit utilizado en el signo.

Como se puede ver, la misma estrategia utilizada para la resolución de los casos de `div` y `rem` pudo ser utilizada para generar los chequeadores de

los operadores de *Shift*. Lo único que cambió fue la algoritmia involucrada dentro del chequeador del comportamiento.

Es así como este método puede ser además utilizado casi sin alteraciones para los casos de comportamientos indefinidos de: verificar que un arreglo no reciba tamaño menor que cero en su inicialización (`chk_neg`), verificar que la diferencia de punteros esté en el rango adecuado (`chk_ptrdiff`, hasta antes de la implementación de punteros *just-beyond*; ver sección 3.3.5), chequear la desreferencia de objetos inválidos (`chk_deref`) y comprobar la generación de ceros negativos (`chk_bitwise_op`).

Información sobre objetos en memoria

Para la detección de algunos comportamientos indefinidos fue necesario extender la técnica usada previamente. Tómese por ejemplo la siguiente definición de un comportamiento indefinido [8, Apéndice J.2, p. 494]:

> Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object.

En el siguiente fragmento de código C donde se declara un arreglo, un puntero a dicho arreglo y luego se realiza una operación de aritmética de punteros, se produce un comportamiento indefinido como el de la definición anterior:

```
int a[5] = {0, 1, 2, 3, 4};
int *b = a;
b = b + 10;
```

Más específicamente, el comportamiento indefinido ocurre en la operación de aritmética de punteros `b + 10`: la suma de un puntero a un arreglo de enteros y una constante entera *mayor que el tamaño del arreglo* cuyo resultado *claramente* no apunta dentro del mismo arreglo (*"Addition [...] of a pointer into [...] an array object and an integer type produces a result that does not point into [...] the same array object."*).

La dificultad para la detección del comportamiento indefinido de este ejemplo está en el hecho de que se necesita conocer el tamaño del arreglo cuando se emite el código *LLVM IR* correspondiente a `b + 10`, que es cuando `idbcc` emite el código *LLVM IR* que llama a la función del entorno `idbre` encargada de detectarlo.

Pero el tamaño del arreglo no está en `b + 10` y por la forma en que está estructurado *Clang* tampoco está en el método de *CodeGen* encargado de emitir el código *LLVM IR* de la aritmética de punteros, que sólo tiene acceso a los operandos; que son objetos de *C++* que representan el código *LLVM IR* correspondiente al puntero a entero `b` y la constante entera `10`.

Nuevamente, por la forma en que está estructurado *Clang*, el tamaño del arreglo `a` solamente está disponible en el método de *CodeGen* que emite el código *LLVM IR* correspondiente a `int a[5]`. Luego, dado que tanto para *LLVM IR* como para *C* los arreglos no son más que un segmento contiguo de memoria, no existe ningún método para averiguar su tamaño una vez creados.

Sin conocer el tamaño del arreglo `a` es imposible detectar el comportamiento indefinido de este ejemplo en tiempo de ejecución. Además es necesario notar que el ejemplo anterior es sumamente sencillo, con toda la información necesaria disponible en el código, de modo que puede determinarse que ocurre un comportamiento indefinido en tiempo de compilación mediante análisis estático. Un ejemplo donde determinar que ocurre un comportamiento indefinido sólo puede hacerse en tiempo de ejecución es el siguiente código *C* de una función:

```
int * offset(int *p, int m) {
    return p + m;
}
```

En la función `offset` se puede ver una situación donde es imposible determinar a partir del código cercano a la operación de aritmética de punteros si ocurre un comportamiento indefinido. La única información con la que se cuenta es el tipo de los argumentos de la operación `p + m`. No es posible saber el tamaño del arreglo al que apunta `p`, si de hecho apunta a un arreglo. Tampoco es posible saber a qué posición del arreglo apunta `p`. Para poder determinar si ocurre un comportamiento indefinido es necesario conocer en tiempo de ejecución dónde comienza y dónde termina el arreglo apuntado por `p`.

Este comportamiento indefinido pertenece a un conjunto de comportamientos indefinidos relacionados con el acceso a objetos en memoria y la utilización de punteros. El diseño de *C* respecto de estas tareas es a la vez una de las razones de la flexibilidad y el poder de este lenguaje así como el origen de innumerables errores. Con el objeto de prevenir o detectar dichos errores se han creado muchas herramientas [9, 4, 15, 20].

En particular es conveniente mencionar el trabajo detallado en *A Bounds Checking C Compiler* [6] y *Backwards-compatible bounds checking for arrays and pointers in C programs* [7], al que en adelante se llamará *B3C*. En *B3C* se modificó el compilador *GCC* [5] y algunas funciones de la biblioteca estándar de *C* y se creó una biblioteca con las funciones necesarias para detectar errores en el acceso a objetos en memoria y la utilización de punteros en tiempo de ejecución.

B3C tiene un par de puntos que son de interés. En primer lugar, define como errores a varios de los comportamientos indefinidos relacionados con el acceso a objetos en memoria y la utilización de punteros, en particular el que se usa como ejemplo en esta sección, y por lo tanto los detecta aunque no sea la detección de comportamientos indefinidos su objetivo.

En segundo lugar, uno de sus principales objetivos es que los programas compilados con el compilador *B3C* puedan ser usados con bibliotecas que no hayan sido compiladas con este. De esta forma, se pueden detectar errores incluso en programas que dependen de bibliotecas de las que no se tiene su código fuente. Gracias a esto también es posible mejorar el desempeño de la ejecución de un programa compilado con el compilador *B3C*, utilizándolo sólo con las partes del código que se desean examinar.

Para lograr esto, el compilador *B3C* instrumenta el programa compilado de forma que cuando se crea un nuevo objeto se almacena en una estructura de datos la dirección de memoria donde comienza y donde termina. De esta forma, se evita modificar la representación de los objetos en sí, de modo que estos pueden ser utilizados con funciones de bibliotecas que no se puedan o no se desean recompilar, con la desventaja de no poder detectar errores que involucran objetos creados por dichas bibliotecas.

Para la detección de los errores, el compilador *B3C* reemplaza en el programa compilado las instrucciones donde se pueden producir los errores que busca detectar, como la operación de aritmética de punteros de los fragmentos de código usados como ejemplo en esta sección, por llamadas a funciones de la biblioteca *B3C*. Estas funciones reproducen la funcionalidad de las instrucciones que reemplazan pero antes buscan en la estructura de datos el objeto involucrado y controlan si se produce un error. Estas llamadas a funciones y las búsquedas que llevan a cabo tienen un fuerte impacto en el desempeño de la ejecución del programa compilado, por lo que *B3C* dedica mucha atención al desempeño de la estructura de datos elegida y optimizaciones para eliminar llamados a las funciones de la biblioteca *B3C* que sean redundantes o innecesarios.

La técnica usada previamente para la detección de comportamientos

indefinidos en *IDefBehav* que consiste en agregar a los programas compilados llamados a funciones de `idbre` es similar a la usada para la detección de errores en el acceso a objetos en memoria y la utilización de punteros en el compilador *B3C*. La única diferencia está en que en *IDefBehav* se evita reemplazar el código *LLVM IR* original generado por *Clang* y en lugar de eso agregar el llamado a la función de `idbre` justo antes de dicho código. El compilador *B3C* en cambio reemplaza el código original generado por *GCC* por llamados a funciones de la biblioteca *B3C*.

Por esta similitud se decidió avanzar con los siguientes cambios a *IDefBehav*, además de agregar en los programas compilados llamados a funciones del entorno `idbre` en el punto donde pueden ocurrir los comportamientos indefinidos.

Primero, agregar en `idbre` una estructura de datos sencilla para almacenar la información de los objetos en memoria durante la ejecución del programa compilado con `idbcc` y las funciones necesarias para insertar y acceder a dicha información.

Segundo, agregar a `idbcc` el código necesario para recolectar la información de cada objeto creado durante la ejecución de un programa y generar llamados a las funciones de `idbre` encargadas de almacenar dicha información en la estructura de datos del mismo.

El principal objetivo de *IDefBehav* es la detección de la mayor cantidad posible de comportamientos indefinidos durante la ejecución de un programa. El desempeño de la ejecución de los programas compilados con `idbcc` nunca fue un objetivo principal, por lo que no se puso el mismo cuidado en el diseño de la estructura de datos o la optimización de los llamados a funciones como en *B3C*.

El lenguaje C clasifica los objetos en memoria por lo que llama *storage duration*. Esta característica determina el tiempo de vida de un objeto en memoria, que es la parte de la ejecución de un programa durante la cual un objeto existe, tiene una dirección de memoria constante y mantiene el último valor almacenado en él. El *storage duration* de un objeto en memoria puede ser uno de tres: *static*, *automatic* o *allocated* [8, Sección 6.2.4, p. 32].

Los objetos en memoria con *static storage duration* existen durante toda la ejecución de un programa y su valor inicial es establecido antes de que comience dicha ejecución. Los objetos cuyos identificadores son declarados a nivel de archivo o con los modificadores `static` o `extern` tienen *static storage duration* [8, Sección 6.2.4, p. 32].

Aquellos objetos en memoria con *automatic storage duration* existen desde el momento en que la ejecución del programa entra en el *bloque de*

código donde están *declarados*, excepto para los *arreglos de tamaño variable* que existen desde que la ejecución del programa alcanza su *declaración*. En cualquier caso, un objeto con *automatic storage duration* existe hasta que la ejecución del programa sale del *bloque de código* donde está *declarado* y su valor es indeterminado hasta que son *inicializados*. Los objetos cuyos identificadores son declarados como argumentos de funciones o a nivel de bloque y sin los modificadores `static` o `extern` tienen *automatic storage duration* [8, Sección 6.2.4, p. 32].

Finalmente, aquellos objetos en memoria con *allocated storage duration* existen desde el momento en que son *alojados* hasta el momento en que son *desalojados* por llamadas a las funciones de manejo de memoria de la biblioteca estándar de *C* como `malloc` o `free` [8, Sección 7.20.3, p. 313].

Al momento de implementar la recolección de información sobre objetos en memoria en *IDefBehav* se consideró recolectar primero la información de los objetos con *allocated storage duration*, que puede ser más sencillo ya que el tiempo de vida de dichos objetos están determinados por llamadas a funciones concretas de la biblioteca estándar de *C*, que pueden ser modificadas en lugar del compilador [6, 7, 20] o interceptadas cuando se *linkea* el programa compilado.

Sin embargo, el método de trabajo usado en *IDefBehav* es justamente la modificación de un compilador, por lo que recolectar la información de los objetos con *allocated storage duration* pasó a segundo plano.

A diferencia de los objetos con *allocated storage duration*, aquellos con *static storage duration* y *automatic storage duration* tienen asociados un identificador, una variable de un tipo determinado, que debe ser declarado. Dentro de *CodeGen* el módulo encargado de emitir el código *LLVM IR* correspondiente a una declaración es *CGDecl*. En dicho módulo se encuentra definida la siguiente función:

```
void CodeGenFunction::EmitAutoVarDecl(const VarDecl &D) {
  AutoVarEmission emission = EmitAutoVarAlloca(D);
  EmitAutoVarInit(emission);
  EmitAutoVarCleanups(emission);
}
```

Como su nombre indica, esta función emite el código *LLVM IR* para las declaraciones de variables con *automatic storage duration*, aunque sólo aquellas que ocurren en el cuerpo de una función. Las declaraciones de los parámetros de una función, que también tienen *automatic storage duration* son manejadas por otra función de *CGDecl*.

La función toma como único argumento una *referencia* a un objeto de *C++* de tipo *VarDecl*, mediante el cual se obtiene toda la información necesaria. Para esto se agrega una llamada a la función del módulo *RuntimeUtil* de `idbcc` encargada de recolectar dicha información después de la llamada a `EmitAutoVarAlloca(D)`, de modo que el objeto ya haya sido creado y por lo tanto tenga una dirección de memoria:

```
void CodeGenFunction::EmitAutoVarDecl(const VarDecl &D) {
  AutoVarEmission emission = EmitAutoVarAlloca(D);

  IDefCGF &ICGF = static_cast<IDefCGF &>(*this);
  ICGF.emitRegAutoVarDeclCall(D);

  EmitAutoVarInit(emission);
  EmitAutoVarCleanups(emission);
}
```

A diferencia de otras funciones del módulo *RuntimeUtil* de la sección *CodeGen* del compilador `idbcc`, que no tienen mucho más código que el necesario para generar una llamada a función en *LLVM IR*, la función `emitRegAutoVarDeclCall` es bastante compleja y fue desarrollada en varias etapas. A continuación, se analizará la primera versión que recolectaba suficiente información como para hacer posible la detección de un comportamiento indefinido:

```
void IDefCGF::emitRegAutoVarDeclCall(const VarDecl &D) {
  Value *FilePos = emitFileLoc(D.getLocation());
  ASTContext *ASTC = &D.getASTContext();
  // Get the object's memory address.
  Value *Addr = GetAddrOfLocalVar(&D);
  ...
```

`emitRegAutoVarDeclCall` tiene como único argumento la referencia `&D` al objeto de *C++* que representa la declaración de una variable. Lo primero que se hace es generar un *string* con el nombre de archivo, la línea y la columna donde ocurre la declaración en el código fuente que se está compilando. Este *string* será pasado a la función de `idbre` para su uso en los mensajes de error. A continuación se obtiene un puntero a un objeto de *C++* de tipo *ASTContext*. Esto se hace sólo para simplificar el código de `emitRegAutoVarDeclCall`. Luego se obtiene la dirección de memoria del objeto creado por la declaración, el primero de los datos buscados.

```
...
// Initialize element count and element size.
Value *ArrLen = NULL;
Value *ElemSize = NULL;
// Get the object's element count and element size.
if (D.getType()->isConstantArrayType()) {
  const ConstantArrayType *Arr =
↪   ASTC->getAsConstantArrayType(D.getType());
  ArrLen = Builder.getInt32(Arr->getSize().getZExtValue());
  ElemSize =
↪   Builder.getInt32(ASTC->getTypeSize(Arr->getElementType()));
} else if (D.getType()->isVariableArrayType()) {
  const VariableArrayType *Arr =
↪   ASTC->getAsVariableArrayType(D.getType());
  ArrLen = getVLASize(Arr).first;
  ElemSize =
↪   Builder.getInt32(ASTC->getTypeSize(Arr->getElementType()));
} else {
  ArrLen = Builder.getInt32(1);
  ElemSize = Builder.getInt32(ASTC->getTypeSize(D.getType()));
}
...
```

Luego se obtienen la cantidad de elementos del objeto y el tamaño de cada elemento individual. Esto se hace de diferentes formas según el objeto sea un arreglo de tamaño constante o variable, o un único elemento. Para esto se utilizan distintos métodos y objetos de C++ provistos por *Clang* según sea el caso. Sobre estos métodos y objetos se profundizará más adelante. Sin embargo, vale la pena notar en este momento que a pesar de las similitudes que pueden esperarse entre distintos tipos de arreglos, *Clang* no siempre provee el mismo método para averiguar el mismo dato.

```
...
llvm::Type *ArgTypes[] = { Addr->getType(), ElemSize->getType(),
↪   ArrLen->getType(), Int8PtrTy };
llvm::FunctionType *RegAutoVarDeclTy = llvm::FunctionType::get(VoidTy,
↪   ArgTypes, false);
Value *RegAutoVarDecl = CGM.CreateRuntimeFunction(RegAutoVarDeclTy,
↪   "reg_avdecl");
```

```
  Builder.CreateCall4(RegAutoVarDecl, Addr, ElemSize, ArrLen, FilePos);
}
```

Finalmente se genera el llamado a la función del entorno `idbre`, en este caso `reg_avdecl`, de manera similar a como se ha visto antes, pasando como argumentos la información recolectada.

El siguiente es el código *LLVM IR* correspondiente a la declaración `int a[5]` compilada por *Clang*:

```
%a = alloca [5 x i32], align 16
```

Al compilar la misma declaración con `idbcc` se obtiene el siguiente código *LLVM IR*:

```
%a = alloca [5 x i32], align 16
call void ([5 x i32]*, i32, i32, i8*, ...)* @reg_avdecl([5 x i32]* %a,
↪    i32 32, i32 5, i8* getelementptr inbounds ([11 x i8]* @0, i32 0,
↪    i32 0))
```

Como se puede ver, a continuación de la instrucción que aloja un arreglo de cinco enteros de 32 bits (`[5 x i32]`) se agrega un llamado a la función `@reg_avdecl` . A esta se le pasan como argumentos: un puntero al arreglo alojado en la instrucción anterior (`[5 x i32]* %a`), el tamaño en bits de un entero (`i32 32`) y la cantidad de elementos del arreglo (`i32 5`), además de un puntero al *string* generado para su uso en los mensajes de error apuntado por la variable global `@0` .

De esta forma, al momento de ejecutar el programa compilado, la función `reg_avdecl` se encarga de almacenar la información recolectada sobre el nuevo objeto en la estructura de datos de `idbre`:

```
void reg_avdecl(size_t addr, unsigned int element_size, unsigned int
↪    elems, const char *fpos) {
    struct auto_var *vw=NULL;
    // Constructor expects bytes but element_size comes in bits.
    vw = new_var(addr, elems, (size_t) element_size/8);
    cl_pushback(&aotable, vw);
}
```

La información recolectada es almacenada en una nueva instancia de `struct auto_var` y un puntero a esta se guarda en la estructura de datos

`aotable`. Estas estructuras de datos y las funciones que las acompañan se describirán en detalle en el capítulo 4

Con la información sobre el objeto almacenada en `aotable` se tiene disponible información suficiente para detectar exitosamente el comportamiento indefinido mencionado al comienzo de esta sección. Sin embargo, este trabajo no recae en `reg_avdecl`, cuyo fin sólo es almacenar la información que lo hace posible. De la misma forma que se ha descrito para otros comportamientos indefinidos, al método de *Codegen* donde se genera el código *LLVM IR* de la aritmética de punteros se lo ha modificado para que justo antes genere un llamado a una función del entorno `idbre`. Esta es `chk_ptraddsub` y es la encargada de detectar dicho comportamiento indefinido. Para ello se le pasan como parámetros los operandos de la operación de aritmética de punteros, que son la dirección de memoria a la que apunta el puntero y el entero que se le sumará o restará.

```
void chk_ptraddsub(size_t addr, long int offset, const char *fpos) {
    struct auto_var *av=NULL;
    int i=0;
    size_t start_addr=0, end_addr=0;

    while (i < aotable.count) {
        // If there are registered objects 0 < aotable.count.
        av = aotable.vars[(aotable.start + i) % AOT_SIZE];
        start_addr = av->addr;
        end_addr = av->addr + (av->elems * av->element_size);
        if (addr >= start_addr && addr < end_addr) {
            // Break to ensure i < aotable.count if the object was found.
            break;
        }
        ++i;
    }
    if (i < aotable.count && (addr + offset > end_addr || addr + offset
↪     < start_addr)) {
        // The object was found if i < aotable.count.
        // Abort if the resulting address falls within the object.
        fprintf(stderr, "%s: Addition or subtraction produces a result
↪     that does not point into, or just beyond, the same array
↪     object.\n", fpos);
        abort();
```

```
    }
}
```

Durante la ejecución del programa compilado con `idbcc`, antes de que ocurra la operación de aritmética de punteros se llama a `chk_ptraddsub`. Esta comienza a revisar las distintas instancias de `struct auto_var` registradas en `aotable.vars` buscando una tal que la dirección de memoria recibida como parámetro (`size_t addr`) caiga dentro del rango de direcciones ocupadas por el objeto que dicha instancia representa. Si no hay registrado un objeto que coincida, la función termina y el programa continua su ejecución. En caso contrario, se verifica el resultado de la operación de aritmética de punteros (`addr + offset`). Si el resultado está fuera del rango de direcciones de memoria ocupadas por el objeto, se muestra un mensaje de error y se interrumpe la ejecución del programa mediante una llamada a la función `abort`. En caso contrario, la ejecución del programa continúa normalmente.

Es importante notar que esta versión de `chk_ptraddsub` toma como comportamiento indefinido el siguiente caso válido según la definición de comportamiento indefinido dada al comienzo de la sección:

```
int a[5] = {0, 1, 2, 3, 4};
int *b = a;
b = b + 5;
```

Esto se debe a que diferentes objetos en memoria pueden estar alojados unos junto otros de modo que la primera dirección de memoria ocupada por un objeto es la primera dirección justo después o *just beyond* de otro; y es imposible distinguir a qué objeto apunta un puntero con la implementación del entorno `idbre` descrita hasta ahora. El método por el cual se solucionó este inconveniente será discutido en la sección 3.3.5.

Al presentar la función `emitRegAutoVarDeclCall` se dijo que fue desarrollada en varias etapas y que la versión analizada era la primera que recolectaba suficiente información como para hacer posible la detección de un comportamiento indefinido. Concretamente, la información recolectada sólo es consistente para las declaraciones de arreglos *unidimensionales*, ya sean de tamaño constante o de tamaño variable. No ocurre lo mismo para el caso de las declaraciones de arreglos *multidimensionales*.

Información sobre arreglos multidimensionales

En *C*, el tipo arreglo es una forma de tipos derivados o *derived types*, que son construidos a partir de otros tipos. Concretamente, un tipo arreglo es un conjunto no vacío, contiguamente alojado, de un tipo particular de elementos o *element type*. Un tipo arreglo está caracterizado por el tipo y el número de sus elementos [8, Sección 6.2.5, p. 35].

Entonces, dada la declaración `int a[5]` se dice que `a` tiene tipo arreglo de enteros o *array of int*. Además, dada una segunda declaración `int b[6]` también se dice que `b` tiene tipo arreglo de enteros, pero se trata de tipos distintos ya que en ambos casos se trata de arreglos de tamaño constante con tamaños distintos. Sería más preciso decir que `a` tiene tipo arreglo de cinco enteros y `b` tiene tipo arreglo de seis enteros.

Un arreglo es un arreglo *multidimensional*, cuando en su declaración ocurren múltiples especificaciones *"arreglo de"* (`[]`) adyacentes [8, Sección 6.7.5.2, p. 116].

Entonces, en el caso de los arreglos *multidimensionales*, dada la declaración `int a[5][6]` se dice que `a` tiene tipo arreglo de arreglo de enteros, o más concretamente arreglo de cinco arreglos de seis enteros.

Las clases de C++ definidas por *Clang* para representar los tipos de *C*, como la clase `Type` y sus derivadas, también muestran cierta *recursividad* en su modo de uso.

Cuando accedemos al método `getType` de un objeto como la instancia de `VarDecl` al que se tiene acceso como parámetro de la función `emitRegAutoVarDeclCall`, se obtiene una instancia de un objeto de tipo `Type`, o `QualType` si el tipo de *C* correspondiente tiene algún calificador como `const`.

Este objeto de tipo `Type` tiene métodos que permiten acceder a información genérica disponible sin importar el tipo de *C* que representa, así como métodos para determinar a qué subclase de `Type` específica a un tipo particular de *C* pertenece. Algunos de estos métodos son `isConstantArrayType` e `isVariableArrayType`. Sin embargo, antes de acceder a cualquier método que estas subclases agreguen a la clase `Type`, es necesario *castear* el objeto a la subclase apropiada. Para esto se utilizan los métodos `getAsConstantArrayType` y `getAsVariableArrayType`.

Una vez que se tiene acceso a objetos de las subclases de `Type` para arreglos, `ConstantArrayType` y `VariableArrayType`, se tiene acceso a información específica sobre los tipos *"arreglo de"* de *C* que representan, en particular aquella que los caracteriza como son el tipo y la cantidad de sus elementos.

Por lo tanto, cuando en `emitRegAutoVarDeclCall` se obtienen el tamaño del tipo y la cantidad de elementos de un arreglo, si se trata de una declaración como `int a[5][6]`, a `reg_avdecl` se le pasaría 5 como cantidad de elementos y $32 \times 6 = 192$ bits o 24 bytes como tamaño del tipo de los elementos, lo que es correcto dado lo visto sobre arreglos *multidimensionales*, pero no es lo esperado.

Si en cambio se trata de una declaración como `int a[v][v+1]` donde la variable `v` vale 5, a `reg_avdecl` se le pasaría 30 como cantidad de elementos pero 0 como tamaño del tipo. En este caso la cantidad de elementos es la esperada pero no el tamaño del tipo de estos.

El problema del tamaño del tipo de los elementos se podría haber solucionado modificando un poco el código de `emitRegAutoVarDeclCall` para obtener la información de los arreglos de tamaño variable. Sin embargo, solucionar la inconsistencia de la información obtenida entre estos y los arreglos de tamaño constante, declarados con los mismos tamaños de dimensiones, requirió modificaciones mayores que se concentran en la parte que obtiene la cantidad de elementos y el tamaño del tipo de estos:

```
...
// Initialize element count.
Value *ArrLen = Builder.getInt32(1);
// Get the object's element count.
QualType ArrType = D.getType();
while (ArrType->isArrayType()) {
  if (ArrType->isConstantArrayType()) {
    const ConstantArrayType *Arr =
↪    ASTC->getAsConstantArrayType(ArrType);
    Value *Len = Builder.getInt32(Arr->getSize().getZExtValue());
    ArrLen = Builder.CreateNUWMul(ArrLen, Len);
    ArrType = Arr->getElementType();
  } else if (ArrType->isVariableArrayType()) {
    const VariableArrayType *Arr =
↪    ASTC->getAsVariableArrayType(ArrType);
    Value *Len = EmitScalarExpr(Arr->getSizeExpr());
    ArrLen = Builder.CreateNUWMul(ArrLen, Len);
    ArrType = Arr->getElementType();
  }
}
// Get the object's element size.
```

```
Value *ElemSize = Builder.getInt32(ASTC->getTypeSize(ArrType));
...
```

La solución encontrada consistió en iterar sobre el objeto `QualType` devuelto por los métodos `getType` y `getElementType`. Mientras estos sean de alguna subclase de `ArrayType`, se los *castea* a `ConstantArrayType` o `VariableArrayType` según corresponda y se genera código *LLVM IR* que obtiene la cantidad de elementos de la dimensión a la que corresponden.

Para obtener la cantidad total de elementos de un arreglo multidimensional en cada iteración se genera código *LLVM IR* que multiplica la cantidad de elementos de la dimensión actual por la productoria de la cantidad de elementos de las dimensiones anteriores.

Finalmente, dado que en la última iteración el método `getElementType` no devuelve un objeto `QualType` que sea de una subclase de `ArrayType`, entonces este objeto representa el tipo de los elementos del arreglo y se lo usa para averiguar su tamaño mediante un llamado al método `getTypeSize`.

El siguiente par de declaraciones de *C* corresponde a uno de los casos en los que la primera versión `emitRegAutoVarDeclCall` no producía los resultados deseados:

```
int v = 6;
int a[5][v];
```

El siguiente es el código *LLVM IR* emitido por una versión sin modificar de *Clang* para las declaraciones anteriores:

```
%v = alloca i32, align 4
%saved_stack = alloca i8*
store i32 6, i32* %v, align 4
%0 = load i32* %v, align 4
%1 = zext i32 %0 to i64
%2 = call i8* @llvm.stacksave()
store i8* %2, i8** %saved_stack
%3 = mul nuw i64 5, %1
%vla = alloca i32, i64 %3, align 16
%4 = load i8** %saved_stack
call void @llvm.stackrestore(i8* %4)
```

A continuación está el código *LLVM IR* correspondiente a las mismas declaraciones emitido por la versión de `idbcc` que tiene la nueva versión de `emitRegAutoVarDeclCall`:

```
%v = alloca i32, align 4
%saved_stack = alloca i8*
call void (i32*, i32, i32, i8*, ...)* @reg_avdecl(i32* %v, i32 32, i32
↪    1, i8* getelementptr inbounds ([11 x i8]* @0, i32 0, i32 0))
store i32 6, i32* %v, align 4
%0 = load i32* %v, align 4
%1 = zext i32 %0 to i64
%2 = call i8* @llvm.stacksave()
store i8* %2, i8** %saved_stack
%3 = mul nuw i64 5, %1
%vla = alloca i32, i64 %3, align 16
%4 = load i32* %v, align 4
%5 = mul nuw i32 5, %4
call void (i32*, i32, i32, i8*, ...)* @reg_avdecl(i32* %vla, i32 32, i32
↪    %5, i8* getelementptr inbounds ([12 x i8]* @1, i32 0, i32 0))
%6 = load i8** %saved_stack
call void @llvm.stackrestore(i8* %6)
```

Sólo se agregan cuatro líneas de código. Primero, la llamada a la función `@reg_avdecl` que registra la información correspondiente a la variable `%v` y luego el siguiente fragmento:

```
%4 = load i32* %v, align 4
%5 = mul nuw i32 5, %4
call void (i32*, i32, i32, i8*, ...)* @reg_avdecl(i32* %vla, i32 32, i32
↪    %5, i8* getelementptr inbounds ([12 x i8]* @1, i32 0, i32 0))
```

Este fragmento se corresponde a las instrucciones emitidas por la función `emitRegAutoVarDeclCall` para obtener el tamaño de la segunda dimensión, su multiplicación con el tamaño de la primera, que es una constante, y finalmente la llamada `@reg_avdecl` que registra la información correspondiente al arreglo `%vla`.

La versión definitiva de `emitRegAutoVarDeclCall` está motivada por el siguiente comportamiento indefinido:

> An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression a[1][7] given the declaration int a[4][5]).

Aquí se tiene nuevamente un comportamiento indefinido para el cual la característica de los arreglos en *C* de no ser más que una secuencia de

objetos alojados contiguamente, imposibilita la detección si no se cuenta con información extra. En este caso, se trata del acceso fuera de rango de un objeto mediante una expresión de subíndice de arreglo, o *array subscript expression*.

Esto se puede dar de dos formas. En la primera, se accede a un objeto fuera de un arreglo a través de un identificador asociado a este, como se puede ver en el siguiente ejemplo:

```
int a[5] = {0, 1, 2, 3, 4};
a[10] = 0;
```

Es evidente que el objeto accedido (`a[10]`), de existir, no pertenece al arreglo `a`. También es evidente que la información recolectada mediante el código *LLVM IR* generado por `emitRegAutoVarDeclCall` es suficiente para esta forma del comportamiento indefinido.

En la segunda forma en que se puede dar este comportamiento indefinido se accede a uno de los elementos del arreglo, pero la expresión que se usa para ello no respeta los tamaños de las dimensiones con que se definió dicho arreglo:

```
int a[5][6] = {{0, 1, 2, 3, 4, 5}, ... };
a[1][10] = 0;
```

El arreglo `a` está definido con 30 elementos y la expresión con la que se lo accede (`a[1][10]`) corresponde su decimoséptimo elemento. Sin embargo la expresión correcta para acceder a dicho elemento es `a[2][4]`, de otra forma el comportamiento es indefinido.

Para detectar la segunda forma de este comportamiento indefinido fue necesario modificar una vez más la función `emitRegAutoVarDeclCall` de modo que pueda generar el código *LLVM IR* necesario para pasarle a la función `reg_avdecl` del entorno `idbre` los tamaños de cada dimensión de los arreglos que se declaren en un programa.

En la última versión de `emitRegAutoVarDeclCall` se vio que para obtener la cantidad total de elementos de un arreglo se generaba código que multiplicaba el tamaño de las dimensiones. De manera similar, en la última versión de esta función que se verá a continuación, se generará código *LLVM IR* que aloje un arreglo de tamaño apropiado y se almacene el tamaño de cada dimensión en él para después pasarlo como parámetro a `reg_avdecl`:

```
void IDefCGF::emitRegAutoVarDeclCall(const VarDecl &D) {
  Value *FilePos = emitFileLoc(D.getLocation());
  ASTContext *ASTC = &D.getASTContext();
  // Get the object's memory address.
  Value *Addr = GetAddrOfLocalVar(&D);
  ...
```

Nuevamente, la primera parte de la función no tiene cambios.

```
  ...
  // Count the number of array dimensions.
  QualType ArrType = D.getType();
  unsigned Dims = 0;
  while (ArrType->isArrayType()) {
    const ArrayType *Arr = ASTC->getAsArrayType(ArrType);
    ArrType = Arr->getElementType();
    Dims++;
  }
  // Reset ArrType.
  ArrType = D.getType();
  ...
```

Para alojar un arreglo donde almacenar los tamaños de las dimensiones de un arreglo declarado por un programa es necesario conocer el tamaño que deberá tener, por lo que el primer cambio en esta versión de `emitRegAutoVarDeclCall` es el agregado de código para contar la cantidad de dimensiones. El método en que se lleva a cabo esta tarea es análogo al usado para calcular la cantidad de elementos en la versión anterior.

```
  ...
  llvm::AllocaInst *DimsArr;
  Value *Ptr;
  unsigned Idx = 0;
  if (Dims > 0) {
    // Create the array of sizes to pass in the generated function call.
    // Store the number of dimensions in the first array position.
    DimsArr = Builder.CreateAlloca(Int32Ty, Builder.getInt32(Dims + 1));
    Ptr = Builder.CreateGEP(DimsArr, Builder.getInt32(Idx));
    Builder.CreateStore(Builder.getInt32(Dims), Ptr);
    ...
```

El siguiente paso es crear el arreglo donde se almacenarán los tamaños de las dimensiones, pero sólo en caso de que el objeto del cual se está obteniendo información se trate de un arreglo. El espacio alojado para almacenar los tamaños es mayor tamaño que el necesario (`Dims + 1`). El lugar extra se usa para almacenar la cantidad de dimensiones.

En particular se debe notar el uso del método `CreateGEP`. Este método crea un puntero a un elemento dado de un objeto, ya sean arreglos o estructuras, y es un requerimiento para generar lecturas o escrituras en estos.

```
...
// Collect the size of each dimension.
while (ArrType->isArrayType()) {
  // Generate pointer to the next position of the array.
  Idx++;
  Ptr = Builder.CreateGEP(DimsArr, Builder.getInt32(Idx));
  if (ArrType->isConstantArrayType()) {
    const ConstantArrayType *Arr =
↪   ASTC->getAsConstantArrayType(ArrType);
    Value *Len = Builder.getInt32(Arr->getSize().getZExtValue());
    Builder.CreateStore(Len, Ptr);
    ArrType = Arr->getElementType();
  } else if (ArrType->isVariableArrayType()) {
    const VariableArrayType *Arr =
↪   ASTC->getAsVariableArrayType(ArrType);
    Value *Len = EmitScalarExpr(Arr->getSizeExpr());
    Builder.CreateStore(Len, Ptr);
    ArrType = Arr->getElementType();
  }
}
// Reset pointer to the first element of the array.
Ptr = Builder.CreateGEP(DimsArr, Builder.getInt32(0));
...
```

A continuación se recolecta el tamaño de cada dimensión. El código de esta parte de la nueva versión de `emitRegAutoVarDeclCall` es muy similar al de la versión anterior. Las diferencias están en la creación de punteros a las sucesivas posiciones del arreglo y el reemplazo de la generación de código para realizar multiplicaciones, por código que almacena los tamaños de

las dimensiones. Finalmente se genera un puntero a la primera posición del arreglo. Este será el que se pasará como parámetro en el llamado a función generado por `emitRegAutoVarDeclCall`.

```
    ...
  } else {
    // For single objects pass a pointer to a single int in the generated
↪     function call.
    DimsArr = Builder.CreateAlloca(Int32Ty, Builder.getInt32(1));
    Ptr = Builder.CreateGEP(DimsArr, Builder.getInt32(Idx));
    Builder.CreateStore(Builder.getInt32(Dims), Ptr);
  }
  ...
```

Si el objeto del que se está recolectando información no se trata de un arreglo sino de un objeto único, se aloja un único entero, se almacena en él el número de dimensiones (0) y se pasa un puntero a este entero en el llamado a función generado.

```
  ...
  // Get the object's element size.
  Value *ElemSize = Builder.getInt32(ASTC->getTypeSize(ArrType));

  llvm::Type *ArgTypes[] = { Addr->getType(), ElemSize->getType(),
↪    Ptr->getType(), Int8PtrTy };
  llvm::FunctionType *RegAutoVarDeclTy = llvm::FunctionType::get(VoidTy,
↪    ArgTypes, false);
  Value *RegAutoVarDecl = CGM.CreateRuntimeFunction(RegAutoVarDeclTy,
↪    "reg_avdecl");
  Builder.CreateCall4(RegAutoVarDecl, Addr, ElemSize, Ptr, FilePos);
}
```

La última parte de esta versión de `emitRegAutoVarDeclCall` es prácticamente igual a la versión anterior salvo por que se pasa como uno de los parámetros un puntero al arreglo creado para almacenar los tamaños de las dimensiones en lugar de la cantidad de elementos.

Nuevamente tómese el siguiente par de declaraciones como ejemplo:

```
int v = 6;
int a[5][v];
```

El código *LLVM IR* generado por `idbcc` con está última versión de `emitRegAutoVarDeclCall` es el siguiente:

```
%v = alloca i32, align 4
%saved_stack = alloca i8*
%0 = alloca i32
%1 = getelementptr i32* %0, i32 0
store i32 0, i32* %1
call void (i32*, i32, i32*, i8*, ...)* @reg_avdecl(i32* %v, i32 32, i32*
↪    %1, i8* getelementptr inbounds ([11 x i8]* @0, i32 0, i32 0))
store i32 6, i32* %v, align 4
%2 = load i32* %v, align 4
%3 = zext i32 %2 to i64
%4 = call i8* @llvm.stacksave()
store i8* %4, i8** %saved_stack
%5 = mul nuw i64 5, %3
%vla = alloca i32, i64 %5, align 16
%6 = alloca i32, i32 3
%7 = getelementptr i32* %6, i32 0
store i32 2, i32* %7
%8 = getelementptr i32* %6, i32 1
store i32 5, i32* %8
%9 = getelementptr i32* %6, i32 2
%10 = load i32* %v, align 4
store i32 %10, i32* %9
%11 = getelementptr i32* %6, i32 0
call void (i32*, i32, i32*, i8*, ...)* @reg_avdecl(i32* %vla, i32 32,
↪    i32* %11, i8* getelementptr inbounds ([12 x i8]* @1, i32 0, i32
↪    0))
%12 = load i8** %saved_stack
call void @llvm.stackrestore(i8* %12)
```

En el siguiente fragmento, que ocurre justo antes de la primera llamada a `@reg_avdecl`, se puede ver lo que ocurre en el caso de que el objeto a registrar no sea un arreglo. Primero se aloja un único entero, luego se genera el puntero que se usará para accederlo y que también se pasará a `@reg_avdecl` y finalmente se almacena el valor 0:

```
%0 = alloca i32
%1 = getelementptr i32* %0, i32 0
store i32 0, i32* %1
```

En el siguiente fragmento, que ocurre justo antes de la segunda llamada a `@reg_avdecl`, se puede ver lo que ocurre en el caso de que el objeto a registrar sea un arreglo. Primero se aloja el arreglo que se pasará a `@reg_avdecl`; se genera un puntero a su primera posición y se almacena la cantidad de dimensiones; se genera un nuevo puntero y se almacena el tamaño de la primera dimensión; se genera un tercer puntero y se almacena el tamaño de la segunda dimensión, que por ser variable requiere primero la lectura del valor de `%v`; y finalmente se genera el valor que se pasará a `@reg_avdecl`:

```
%6 = alloca i32, i32 3
%7 = getelementptr i32* %6, i32 0
store i32 2, i32* %7
%8 = getelementptr i32* %6, i32 1
store i32 5, i32* %8
%9 = getelementptr i32* %6, i32 2
%10 = load i32* %v, align 4
store i32 %10, i32* %9
%11 = getelementptr i32* %6, i32 0
```

Con esta versión de `emitRegAutoVarDeclCall` se genera el código *LLVM IR* que recolecta la información necesaria que permite detectar los dos comportamientos indefinidos introducidos en esta sección.

Como se puede ver, la forma de lograr este objetivo tuvo el efecto indeseado de que el código *LLVM IR* generado por `idbcc` difiere del generado por *Clang* mucho más de lo que se pretendía inicialmente. Más aún, las diferencias son proporcionales a los tamaños de los arreglos que se declaren en el programa a compilar. Esto hace que sea más difícil de comparar con el código original y verificar que la semántica del programa compilado con `idbcc` sea fiel a la original.

Hasta este punto en el desarrollo del trabajo se había encarado cada nuevo comportamiento indefinido con la intención de minimizar los cambios en el código *LLVM IR*. Esto dejó de ser cierto con el código que se genera para recoger la información necesaria para la base de datos de objetos. Sin embargo, nunca se había alterado el código que *LLVM* ya generaba hasta que se llegó a los comportamientos indefinidos tratados en la sección 3.3.5.

Los punteros Just-Beyond

La dirección *just-beyond* es la dirección de memoria que sucede a los objetos de tipo *array* (arreglo), es decir, que la posición *just-beyond* es exclusivamente relativa a un objeto de tipo *array*. Esta dirección es especial y no contiene ningún elemento del arreglo al que pertenece. Un puntero apuntando a dicha posición de memoria es lo que se denominará en esta sección como un *puntero a just-beyond*.

Como tal, un puntero a just-beyond tiene algunas características: puede ser utilizado en las operaciones aritméticas de punteros al objeto que pertenece, pero así también, no puede ser *desreferenciado* para intentar leer el valor que contiene.

En la escritura de programas en *C* es muy común obtener este tipo de punteros ya que es el valor que generalmente termina adoptando la variable de iteración cuando se recorren todos los elementos de un arreglo, cabe aclarar, siempre que se hayan usando directamente punteros al objeto para la iteración en lugar de un representante numérico de la posición relativa de las direcciones de memoria dentro del objeto.

Por ejemplo:

```
void foo(void) {
    int p[5];
    int *q=NULL;

    for (q=p; q<p+5; ++q) ;
}
```

es un caso muy simple de código que alcanza ese objetivo. El incremento de `++q` mientras este esté por debajo `p+5`, va a terminar de ciclar exactamente cuando `q` valga `p+5`, quedando así con el valor a la dirección de memoria consecutiva al objeto apuntado por `p`. El `q` así construido es un puntero a una dirección de memoria just-beyond, un puntero a just-beyond.

Una cuestión no menor de este tipo de punteros es que antes de adentrarse en el análisis de los comportamientos indefinidos que puedan generar, hay que poder distinguirlos bien de punteros ordinarios a direcciones de memoria comunes y corrientes. Por ejemplo, en un programa podría ocurrir la declaración de más de un objeto y que el sistema operativo subyacente opte por almacenar estos objetos de forma consecutiva, uno detrás del otro. En dicho caso, como se puede ver en la figura 3.2b, la dirección *just beyond* del primer objeto en la memoria estaría apuntando a

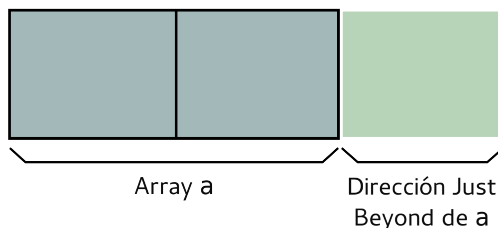
la dirección inicial del segundo, generando la ambigüedad de si el puntero en cuestión es o no un puntero just-beyond.

A modo de ejemplo, supóngase que no se disponga *marca* sobre un puntero just-beyond, y supóngase también que en un programa se declaran 2 objetos de tipo arreglo como en el siguiente programa de ejemplo:

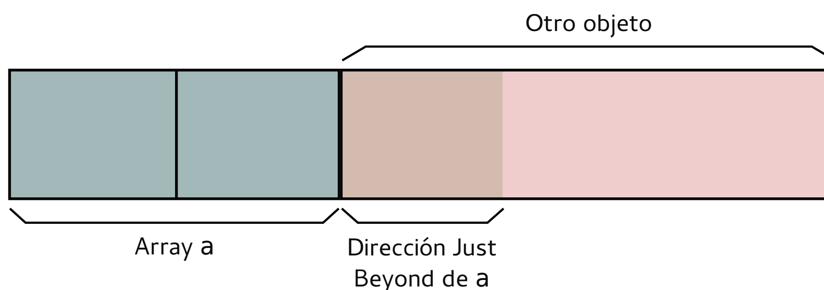
```
int main(void) {
    int p[2];
    int q[2];
    printf("%ld\n", p - (q+2));
    return 0;
}
```

es fácil ver que el resultado impreso va a ser en varios casos, y dependiendo del compilador, `0`, lo cual quiere decir que el valor just-beyond del arreglo `q` es efectivamente la dirección del objeto del arreglo `p`. Cabe aclarar que no hace falta ningún flag de compilación raro en absoluto:

```
$ clang pointer_sub.c
$ ./a.out
0
```



(a) Dirección just beyond del arreglo `a`



(b) Superposición de la dirección just beyond de `a` con el espacio de memoria de otro objeto

Figura 3.2: Just Beyond

Es cierto que esta peculiaridad es gentileza del compilador, y que podría no haberse dado tanto como haberse dado en el orden inverso.

Una estrategia utilizada en herramientas de *debugging* es el uso de espacio de *padding* para separar los objetos en la memoria [6, 7, 20], asegurando de esta forma que, al 2 objetos estar lo suficientemente separados entre sí, ningún espacio de la memoria de uno, o relativo a uno de ellos como es el caso de la dirección just-beyond, se encuentre en el adyacente espacio de memoria de otro objeto provocando ambigüedades.

En este trabajo se optó por una segunda opción, equivalente a lo que se hizo en la sección 3.3.3. Para llevar el registro de las direcciones de memoria just-beyond, más precisamente de los punteros *efectivamente creados* a este tipo de direcciones, se generó una estructura en memoria que lleva cuenta de todos los punteros construidos a direcciones just-beyond y los marca.

Marcar estos objetos consiste en generar una estructura que almacena, de la dirección apuntada por el puntero just-beyond, la dirección en cuestión, el objeto *padre* del que resultare la dirección just-beyond y la cantidad de veces que esta dirección de memoria es apuntada. Una vez almacenada esta información, se devuelve, para ser almacenada en la referencia donde se guarda el valor del puntero, la dirección en el registro de este nuevo objeto.

Es importante recalcar que, la única forma posible de obtener un puntero just-beyond es a partir de la operación de suma de punteros. No es posible otra forma dado que la dirección de memoria just-beyond de un arreglo, se define como tal en base al objeto arreglo del que forma parte. Si bien su dirección puede coincidir con la de otro objeto de la memoria, un puntero a este último objeto es un puntero común y corriente dado que, por su construcción es un puntero al comienzo de un objeto. De esta manera, se hace la distinción explícita de que un puntero just-beyond tiene que ser obtenido a partir del objeto del que forma parte.

Habiendo descartado así toda otra forma de generación de los punteros just-beyond, salvo la suma de punteros partiendo de direcciones dentro del objeto del que es parte, resulta fácil detectar cuándo es necesario realizar la operación de almacenamiento, registro y conteo de punteros just-beyond. Basta con interceptar la generación de instrucciones de la forma `getelementptr inbounds <type>* %array, <type> %value` y reemplazar allí por el llamado al chequeador en el entorno `idbre`. Estos llamados en el *IR* de *LLVM* pueden, a su vez, ser generados en dos puntos diferentes en el compilador: uno de los puntos es en el momento de reali-

zar sumas de punteros, tal como la aplicada por el operador de suma +, y el otro punto es computando sobre el puntero un incremento o decremento utilizando los operadores unarios `++` y `--`.

De esta forma, se logró entonces que en vez de generarse:

```
for.cond:                                         ; preds = %for.inc,
↪    %entry
  %0 = load i32** %q, align 8
  %arraydecay1 = getelementptr inbounds [5 x i32]* %p, i32 0, i32 0
  %add.ptr = getelementptr inbounds i32* %arraydecay1, i64 5
  %cmp = icmp ult i32* %0, %add.ptr
  br i1 %cmp, label %for.body, label %for.end
```

o:

```
for.inc:                                          ; preds = %for.body
  %1 = load i32** %q, align 8
  %incdec.ptr = getelementptr inbounds i32* %1, i32 1
  store i32* %incdec.ptr, i32** %q, align 8
  br label %for.cond
```

se generen sus bloques respectivos:

```
for.cond:                                         ; preds = %for.inc,
↪    %entry
  %6 = load i32** %q, align 8
  %arraydecay1 = getelementptr inbounds [5 x i32]* %p, i32 0, i32 0
  %arraydecay2 = getelementptr inbounds [5 x i32]* %p, i32 0, i32 0
  %add.ptr.idb = call i32* (i32*, i64, i8*, ...)* @chk_ptraddsub(i32*
↪    %arraydecay2, i64 20, i8* getelementptr inbounds ([12 x i8]* @3,
↪    i32 0, i32 0))
  call void (i32*, i32*, i8*, ...)* @cmp_ptr_parents(i32* %6, i32*
↪    %add.ptr.idb, i8* getelementptr inbounds ([12 x i8]* @4, i32 0,
↪    i32 0))
  %7 = call i32* (i32*, i8*, ...)* @get_real_address(i32* %6, i8*
↪    getelementptr inbounds ([12 x i8]* @5, i32 0, i32 0))
  %8 = call i32* (i32*, i8*, ...)* @get_real_address(i32* %add.ptr.idb,
↪    i8* getelementptr inbounds ([12 x i8]* @5, i32 0, i32 0))
  %cmp.idb = icmp ult i32* %7, %8
  br i1 %cmp.idb, label %for.body, label %for.end
```

y:

```
for.inc:                                            ; preds = %for.body
  %9 = load i32** %q, align 8
  %incdec.ptr.idb = call i32* (i32*, i64, i8*, ...)* @chk_ptraddsub(i32*
↪    %9, i64 4, i8* getelementptr inbounds ([12 x i8]* @6, i32 0, i32
↪    0))
  store i32* %incdec.ptr.idb, i32** %q, align 8
  br label %for.cond
```

Cabe aclarar que no todas las generaciones de `getelementptr inbounds` en el *IR* interesan ser atrapadas. Afortunadamente, los extractos de *IR* expuestos dan una pista: sólo interesan los que son marcados con la secuencia de caracteres `%add.ptr` y `%incdec.ptr` en el código del compilador.

La diferencia principal con los casos tratados en las secciones previas, es que para todos los casos expuestos hasta ahora en este trabajo, siempre bastó con anteponer el llamado al chequeador, a la operación que se deseaba verificar para controlar a partir de los argumentos que la operación se pueda realizar de manera segura. Con el procedimiento mencionado que se utilizó para este tipo de objetos en vez, hay que reemplazar por completo el generador de la suma tal como se vio en los ejemplos de código previos, donde se reemplaza por completo la instrucción `getelementptr inbounds ...` por `call ... @chk_ptraddsub(...)` en todos los casos. Esto tiene que ser de esta forma debido a que la marca del puntero just-beyond se realiza registrando su información en una tabla separada en otra ubicación de la memoria; y lo que se quiere devolver como resultado de la suma es la dirección *en la tabla* del puntero guardado.

De esta forma, y a modo de resumen de lo expuesto hasta el momento, cuando se computa una suma de punteros que resulta en una dirección de memoria just-beyond, se elige el siguiente lugar vacío en la tabla que registra estos elementos, y la operación de suma devuelve la dirección real, dentro de la tabla auxiliar, de este nuevo objeto creado.

De esta manera, el objeto que computó para su almacenamiento una dirección just-beyond se lleva consigo una dirección única en memoria que se encuentra en un rango de direcciones manejados por la herramienta de *runtime* y que si hubiera coincidido con la dirección de comienzo de otro objeto en la memoria ahora va a ser, por su forma de cómputo, inambiguamente diferente. De esta manera, comprobar si la dirección de memoria es o no un puntero just-beyond consiste en verificar que la dirección se encuentra en el rango de memoria reservado para las tablas de estos objetos, y de estarlo allí, es trivial la toma de decisiones respecto

a comportamientos indefinidos que involucren direcciones de memoria just-beyond.

La dificultad de este enfoque es que se está guardando, para uso en todas las operaciones posteriores a su construcción, una dirección que a diferencia de lo esperado, no es consecutiva al objeto del que es derivada. De manera que hay que capturar varias operaciones más para garantizar la consistencia de los programas que hagan usos legales de este tipo de direcciones.

A modo de ejemplo, poder restar un puntero a un puntero just-beyond, cuyo resultado termine en el rango de memoria del objeto al que pertenece el puntero just-beyond, es una operación completamente válida; comparar el puntero just-beyond con otros punteros a direcciones de memoria dentro del objeto al que el just-beyond pertenece, también es completamente válido. De manera que a la hora de realizar estas operaciones es necesario restaurar el valor del puntero just-beyond original para ser pasado a este tipo de operaciones aritméticas y lógicas.

Los comportamientos indefinidos

Una vez alcanzadas todas estas consideraciones previas, se procedió a la detección y verificación de los comportamientos indefinidos que involucran a los punteros just-beyond.

Se encuentran listados en el estándar los siguientes comportamientos indefinidos al respecto:

> Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object.

que dice que es un comportamiento indefinido si se suma o resta un puntero adentro de un objeto de tipo arreglo, o su dirección just-beyond, con algún tipo entero quedando el resultado fuera del rango de memoria del objeto, o su dirección just-beyond.

Para el tratado de este comportamiento indefinido ya fue necesaria la utilización de toda la infraestructura escrita para el manejo de punteros just-beyond. Se puede ver más o menos fácilmente que para la suma o resta partiendo de un puntero a cualquier dirección del objeto, salvo la dirección just-beyond, basta con toda la infraestructura construida para la sección 3.3.3, donde basta con preguntar si el resultado de la suma o la resta caen dentro de los límites deseados. Pero si el puntero participando de la

suma o resta es el puntero just-beyond, allí es donde viene la necesidad de lo aquí construido.

Si se sabe que el puntero del que se origina la operación es un puntero just-beyond, éste tiene en su registro, además de su dirección de memoria original, la del objeto al que pertenece. Entonces, controlar que al sumar o restar un puntero just-beyond a un entero, cumpla con que el resultado de la operación quede en el rango del objeto padre, es prácticamente igual de sencillo que si hubiera sido un puntero cualquiera a una dirección dentro del objeto padre. El chequeador en cuestión resulta siendo el siguiente:

```
if ((void *) jbtable.vars > (void *) addr ||
    (void *) addr > (void *) &(jbtable.vars)[N-1]) {
    while (i < aotable.count) {
        av = aotable.vars[(aotable.start + i) % N];
        start_addr = av->addr;
        end_addr = av->addr + (av->elems * av->element_size);
        if (addr >= start_addr && addr < end_addr)
            // Break before increment assuring that i is never going
            // to reach aotable.count if a correct object is found.
            break;
        ++i;
    }
if (i != aotable.count  // An object has been found.
    && (addr + offset > end_addr || addr + offset < start_addr)) {
    abort();
}
```

Un segundo comportamiento involucrando punteros just-beyond, que este trabajo verifica es:

> Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated.

Las condiciones para llegar a este caso son similares al comportamiento anterior, pero en este caso si el resultado de la operación aritmética es la propia dirección just-beyond, y el puntero a esta dirección es utilizado con el operador unario de des-referencia *, es un comportamiento indefinido.

Este caso es bastante más sencillo que el anterior. Basta con interceptar el fragmento de código en Clang donde se genera la des-referencia e insertar allí la generación del llamado al chequeador, para cualquier puntero a ser des-referenciado:

```
Value *VisitUnaryDeref(const UnaryOperator *E) {
  IDefCGF &ICGF = static_cast<IDefCGF &>(CGF);
  if (E->getSubExpr()->getType()->isPointerType())
    ICGF.emitChkDerefCall(E->getSubExpr(), E->getOperatorLoc());
  if (E->getType()->isVoidType())
    return Visit(E->getSubExpr()); // the actual value should be
↪    unused
  return EmitLoadOfLValue(E);
}
```

Una vez generado ese llamado, el chequeador sólo tiene que verificar que la dirección que recibió se encuentra dentro del rango de memoria donde está almacenada la tabla de los punteros just-beyond, y si lo encuentra abortará inmediatamente:

```
...
} else if ((void *) jbtable.vars <= ptr &&
           (void *) &(jbtable.vars)[N-1] >= ptr) {
    fprintf(stderr, "%s: Operand of * is a just-beyond address.\n",
            fpos);
    abort();
} else ...
```

Un tercer comportamiento indefinido en el que los punteros just-beyond se ven involucrados, es el siguiente:

> Pointers that do not point into, or just beyond, the same array object are subtracted.

que dice que es comportamiento indefinido si se restan punteros que no apuntan a objetos dentro del mismo arreglo o su correspondiente just beyond.

El chequeador que se construyó para este caso resulta bastante sencillo de pensar debido a toda la instrumentación que se ha dispuesto sobre los objetos en memoria, como se ve en la sección 3.3.3, como así también en

los punteros just-beyond que se vienen describiendo a lo largo de esta sección.

Entonces, para cada puntero involucrado en la operación de resta, el chequeador primero encuentra cuál es la dirección de memoria donde el objeto de tipo arreglo comienza:

```
size_t get_parent(size_t addr) {
    struct auto_var *av=NULL;
    struct just_beyond *addr_p=NULL;
    int i=0;
    size_t start_addr=0, end_addr=0;

    if ((void *) jbtable.vars > (void *) addr ||
        (void *) addr > (void *) &(jbtable.vars)[N-1]) {
        while (i < aotable.count) {
            av = aotable.vars[(aotable.start + i) % N];
            start_addr = av->addr;
            end_addr = av->addr + (av->elems * av->element_size);
            if (addr >= start_addr && addr < end_addr)
                // Break before increment assuring that i is never going
                // to reach aotable.count if a correct object is found.
                break;
            ++i;
        }
    } else {
        // The addressed value is in the just beyond table.
        addr_p = *((void **) addr);
        start_addr = addr_p->parent_object;
    }

    return start_addr;
}
```

siendo `start_addr` la dirección inicio del objeto array correspondiente al puntero por cuya dirección de memoria se está preguntando. Esto mismo es realizado para ambos punteros en la resta.

Si ambas direcciones de memoria, la dirección de comienzo del objeto array que contiene al puntero `p` como la de `q`, no coinciden en ser la misma, entonces el chequeador aborta la ejecución con el mensaje de error correspondiente. Caso contrario la operación continúa su evaluación.

Hay que notar que la parte pertinente en el ejemplo de código correspondiente a esta sección está en el desarrollo de la sentencia `else` de la función `size_t get_parent(size_t addr)` en donde, para la obtención de la dirección del objeto de tipo arreglo al cual la dirección just-beyond pertenece, es preguntada en el campo correspondiente del puntero just-beyond.

Por último, un comportamiento más que es alcanzado por el uso de punteros just-beyond es el siguiente:

> Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators.

Este comportamiento menciona que es indefinido el resultado si se comparan dos punteros que apunten a diferentes objetos *aggregate*, *union* u objetos de tipo arreglo, tanto para sus direcciones internas como para su dirección just-beyond.

Este caso es análogo al caso anterior en lo que respecta a la comparación de punteros just-beyond; se obtiene la dirección del objeto al que pertenencen ambos punteros, con la misma función previamente mencionada y se compara para ver si son el mismo objeto; en caso de que así sea, se continúa con la operación de comparación, en caso contrario se aborta.

Los casos que no involucran a punteros just-beyond, nuevamente fueron resueltos por la infrasetructura explicada en la sección 3.3.3

4 EL ENTORNO DE EJECUCIÓN IDBRE

El entorno de ejecución IDefBehav Runtime Environment *idbre* tiene como objetivo proveer un conjunto de herramientas que realicen la verificación de los comportamientos indefinidos que se desean evaluar, en tiempo de ejecución.

Se eligió armar un entorno de ejecución en lugar de un compilador por varios motivos. Uno de ellos es el hecho de que un compilador sólo puede analizar los comportamientos que ocurran en tiempo de compilación, pudiendo evaluar todo aquello que sea constante y que no cambie a lo largo de la ejecución del programa, pero dejando de lado todos los comportamientos que puedan ocurrir por la interacción del programa con usuarios, otro software y hardware. Si bien varias situaciones pueden ser pre-calculadas en tiempo de compilación, como por ejemplo hacer el seguimiento de los valores de una variable que sólo tome entradas determinadas por la ejecución del programa y nada más, en la mayoría de las ocasiones es la interacción con el medio en que el programa se ejecuta lo que genera las situaciones más interesantes de encontrar en que se generan comportamientos indefinidos.

Segundo, hay muchas herramientas ya que realizan análisis estático de código, entre *debuggers* y otras herramientas de desarrollo, como así mismo los propios compiladores, que ya muchos reportan algunos de los comportamientos indefinidos en forma de "Advertencias" (*Warnings*), si los encuentran a la hora de compilar el programa. Cuando a comienzos de este trabajo se realizó una división entre los casos que se podían resolver de forma estática (*en tiempo de compilación*) y los de forma dinámica (*en tiempo de ejecución*) se encontró que la gran mayoría de los casos estáticos se encontraban en mayor o menor medida cubiertos por uno u otro compilador (GCC, Clang, icc, entre otros).

El entorno de ejecución consta de 2 partes, la primera de ellas, y la más simple desde el punto de vista del entorno de ejecución, se encuentra dentro de la generación de código, más precisamente en la sección *RuntimeUtil* que se puede apreciar en la figura 4.1; que además de encargarse de toda la parte en el compilador de la recolección de datos de los objetos necesarios para realizar los chequeos, es el que realiza la generación del

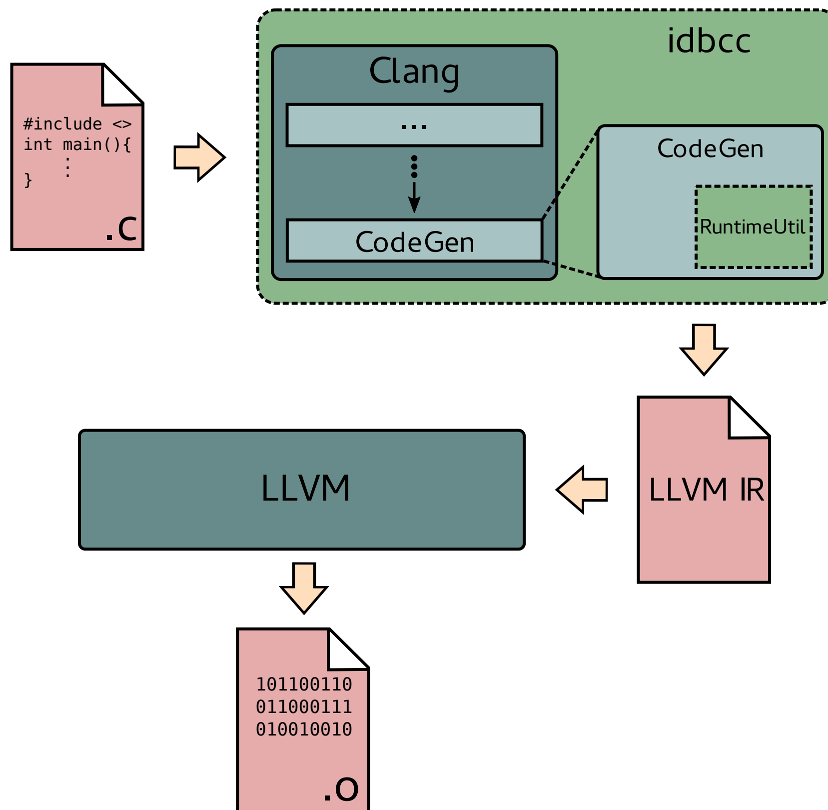


Figura 4.1: Secuencia de compilación

llamado a la función correspondiente entre las bibliotecas de `idbre`. Un ejemplo sencillo de cómo se realiza esta generación es:

```
void IDefCGF::emitChkArraySizeNegCall(Value *Size, const SourceLocation
↪    Loc) {
  llvm::Value *FilePos = emitFileLoc(Loc);
  llvm::Type *ArgTypes[] = { Size->getType(), Int8PtrTy };
  llvm::FunctionType *ChkNegTy =
    llvm::FunctionType::get(VoidTy, ArgTypes, false);
  llvm::Value *ChkNeg = CGM.CreateRuntimeFunction(ChkNegTy, "chk_neg");
  Builder.CreateCall2(ChkNeg, Size, FilePos);
}
```

para construir el llamado a una función con signatura `void chk_neg(int a, const char *fpos)` que se corresponde con una de las funciones dentro de `idbre`.

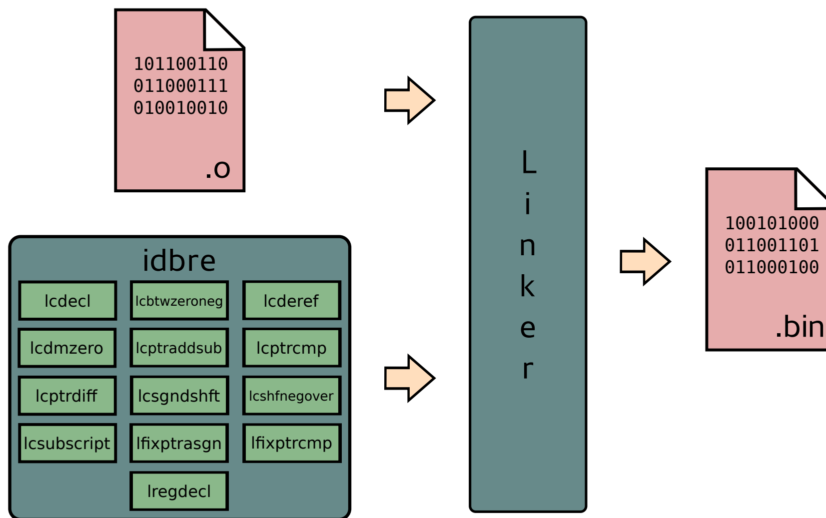


Figura 4.2: Secuencia de *linkeo*

La segunda parte, la principal del entorno `idbre`, es la que se puede observar en la figura 4.2. Esta parte consta de diferentes módulos que pueden ser construidos por separado. Cada módulo posee las funciones chequedoras de un comportamiento indefinido, si un comportamiento indefinido necesita de más de una función chequeadora (mismo chequeo pero con diferente *tipado*) todas esas funciones se ubican en el mismo módulo. Dentro de cada módulo, a su vez se encuentran todas las funciones auxiliares que el verificador del comportamiento indefinido necesite, que no sean necesitadas por otros módulos.

Cada módulo se encuentra escrito en C, por simpleza principalmente, tanto los chequeadores como sus funciones auxiliares. Un ejemplo de uno de estos módulos es:

```
void chk_szero(int a, const char *fpos) {
    // The value of the second operand of the / or % operator is zero
↪    (6.5.5).

    if (a == 0) {
        fprintf(stderr, "%s: Second operand of a / or %% is zero.\n",
↪    fpos);
        abort();
    }
}
```

```
void chk_uzero(unsigned a, const char *fpos) {
    // The value of the second operand of the / or % operator is zero
↪    (6.5.5).

    if (a == 0) {
        fprintf(stderr, "%s: Second operand of a / or %% is zero.\n",
↪    fpos);
        abort();
    }
}

void chk_fzero(double a, const char *fpos) {
    // The value of the second operand of the / or % operator is zero
↪    (6.5.5).

    if (a == 0) {
        fprintf(stderr, "%s: Second operand of a / or %% is zero.\n",
↪    fpos);
        abort();
    }
}
```

Como se puede observar, este es el contenido del módulo encargado de verificar que no se realice el cómputo de una división o resto por cero.

Cada módulo cuenta además con una versión *Stub* de sí mismo, que consta de funciones gemelas que exponen la misma signatura por cada función, pero su contenido es neutral en la ejecución que pase por allí. Esto es, que si una función chequeadora sólo se dedica a realizar una verificación y abortar la ejecución en caso de no pasarla, su versión *Stub* no hará nada, tendrá cuerpo vacío, de manera de no interferir con la ejecución del programa.

Este enfoque se realizó bajo la premisa de que en muchas ocasiones se pretende verificar un comportamiento indefinido específico, pero por la forma de construcción de los ejemplos no se podría obtener un código intermedio *LLVM IR* que contenga sólo el llamado al chequeador deseado, sin anteponer el llamado a otros chequeadores. En estos casos, como se genera el llamado a nivel *IR*, se opta simplemente por *linkear*, a la hora de construir el ejecutable, contra la versión *Stub* de los chequeos que no se desean realizar.

Un claro ejemplo de la utilidad de esta modularización es el verificador del comportamiento indefinido que involucra la generación de *Ceros Negativos*. Este comportamiento indefinido, sólo tiene relevancia si se construye un cero negativo en arquitecturas con problemas para representarlos, no obstante, si no se *linkea* contra la versión *stub* del chequeador, la comprobación se realiza también en, por ejemplo, arquitecturas con representación *complemento 2* de los enteros; limitando el rango de valores válidos que en esta arquitectura se pueden construir, por el simple hecho de que el verificador encontró que se alcanzó un valor inválido para arquitecturas con representación signed magnitude.

Para la construcción de cada módulo se utilizó `gcc` con los siguientes flags: `-std=c99`. Se puede usar cualquier compilador de C salvo `idbcc`.

El motivo por el que se utiliza otro compilador, es porque, si bien el resto del código del runtime no incurre en la generación de los comportamientos que está verificando cada módulo, sí genera las condiciones que originan los llamados a los chequeadores. Al ocurrir esto, puede suceder que un chequeador se termine invocando a sí mismo recursivamente, terminando en un *overflow* del *stack* de llamadas a funciones; como así también necesitar linkearse contra el resto de los módulos en el entorno `idbre`, si bien esto último no es un inconveniente extra gracias a la posibilidad de utilizar las versiones *Stub* de todos los otros módulos contra los que se *linkee*.

El constructor de `idbre` se construye como un compendio de bibliotecas estáticas, existe una biblioteca por módulo chequeador, y a su vez por cáda módulo, su correspondiente biblioteca *Stub*. Además, cada biblioteca contiene el código *objeto* de módulos auxiliares, no chequeadores, que pudiera precisar. Esto es así para no necesitar proveer más de un código objeto al momento del *linkeo* con el programa a verificar y que sea visualmente distinguible cuántos chequeos se le están *linkeando* al programa. De esta forma, módulos como los que llevan el registro de *objetos en memoria*, como *punteros just-beyond*, son empaquetados en la bilbioteca estática del módulo *regdecl*.

El método de utilización del entorno de ejecución es principalmente manual eligiendo qué chequeo se desea realizar y *linkeando* el programa con la biblioteca pertinente. Por ejemplo, si se desea probar el siguiente programa:

```
#define N 2
int main(void) {
```

```
    int p[N], *q=p+N;
    printf("%ld\n", q - p);
}
```

y se lo desea verificar por la realización correcta de la resta de punteros, pero no se desea que la verificación del uso correcto de punteros *just-beyond* entre en juego, se lo compilará de la siguiente manera:

```
$ idbcc -std=c99 ptr-diff.c -L /path/to/runtime/lib -lchkopsptrdiff
↪   -lregdecl -lchkopsptraddsub-stub -lfixopsptrcmp-stub
```

Mientras que si sí se desea verificar el uso correcto de los punteros *just-beyond*, se lo construirá de esta otra manera:

```
$ idbcc -std=c99 ptr-diff.c -L /path/to/runtime/lib -lchkopsptraddsub
↪   -lfixopsptrcmp -lregdecl -lchkopsptrdiff-stub
```

A modo experimental, el entorno de ejecución posee un ejecutable escrito en *python* que arma una batería de combinaciones, de verificaciones, para ir encendiendo y apagando chequeadores (con y sin sus versiones *Stub*) con el objeto de poder automatizar el chequeo de más de un comportamiento indefinido, de un programa arbitrario. Este ejecutable asume un proyecto que pueda construirse con el comando *UNIX* `make` y construye el proyecto modificándole su variable de *linkeo* `LDFLAGS` en cada compilación para elegir con cuáles de los chequeadores será enlazado el binario principal del proyecto en verificación. Una vez construido el binario del programa a verificar, en cada prueba de la batería de tests, lo lanza a correr por un máximo de tiempo fijo y si incurre, antes de la finalización de ese tiempo, en algún comportamiento indefinido lo reporta por pantalla. Tal infraestructura es la que se utilizó para la realización de la prueba en el mundo real detallada en el capítulo 5.

5 PRUEBA DE MUNDO REAL: HTOP

Para controlar la calidad de lo producido se buscó un caso real con el que se pudiera realizar un simulacro de ejecución para ver en cuántos comportamientos indefinidos incurriría un proyecto de software cualquiera.

Entre las alternativas que se evaluaron se buscó algo que estuviera acorde al alcance esperado para un proyecto de tesis de grado. Se evitaron explícitamente piezas de software muy grandes o complejas que no permitieran la correcta comprensión de los resultados por posibles falencias (*bugs*) en el desarrollo del entorno de ejecución. También se analizó la posibilidad de probar el entorno de ejecución con bibliotecas.

Finalmente, el proyecto de software contra el que se probó el entorno de ejecución fue *htop* [14]. *Htop* es un software interactivo para gestión de procesos en sistemas *UNIX* que corre en terminales de texto gráficas y consolas; está diseñado como una alternativa más cómoda a la herramienta de *UNIX* top.

Htop posee algunas características ideales para ser sometido a estas pruebas:

- Es un proyecto de Software Libre al que se le puede acceder el código fuente para compilarlo.
- Está íntegramente escrito en C. Es decir, no se tendrían que compilar algunas partes del software con el compilador aquí explicado y otras partes con otros compiladores.
- Su tamaño es relativamente pequeño (2,9MB source) para una pieza de software, y por consiguiente es más sencillo de recorrer y analizar su código cuando se quiere comprender el origen de sus comportamientos indefinidos.
- Depende de un conjunto muy pequeño de bibliotecas externas a la hora de probarlo, y que podrían interferir con los casos que se desean evaluar.

- Puede correr sin necesidad de interacción de un usuario y su comportamiento no resulta determinístico ya que la interacción ocurre por la generación de otros procesos que se lanzan y mueren continuamente en sistemas *UNIX*. Esta característica lo vuelve mas sencillo de someter a diversos chequeos en menor tiempo.

- Además, es un proyecto ampliamente usado y reconocido por la comunidad.

Para probar *htop* se construyó el *script python* que se menciona en el capítulo 4. El script en cuestión realiza sobre *htop* una totalidad de 12 chequeos diferentes en donde se verifican comportamientos indefinidos por:

- Declaraciones, para verificar el tamaño provisto a arreglos.
- Generación de *ceros negativos*.
- Desreferencia de punteros, tanto en punteros ordinarios como *just-beyond*.
- Cálculo de división y resto por cero.
- Suma de enteros a punteros dentro del rango de direcciones de memoria permitidos.
- Comparación y resta de punteros en los rangos adecuados de direcciones de memoria.
- Cómputo de operaciones de *shift* con valores enteros permitidos.
- Acceso correcto a elementos en arreglos.

Por cada uno de los chequeos a los que *htop* fue sometido se hizo la siguiente rutina: primero se lo compiló corriendo `make` con la asignación en `LDFLAGS` de las bibliotecas del entorno de ejecución contra las que se lo quería chequear, y la asignación a `CFLAGS` del flag de compilación `-std=c99` dado que a lo largo de todo el trabajo siempre se quisieron controlar los comportamientos indefinidos de esa versión del estándar.

Una vez construido el binario se lo ejecutó por un total de 10 segundos, si el proceso no terminaba abortando se lo detenía para proceder a otro test. Se probaron diferentes variantes de configuraciones del entorno `idbre`

donde se lo puso a correr durante más tiempo, hasta 10 minutos de ejecución ininterrumpida (salvo por comportamientos indefinidos), y con mayor espacio de memoria para la creación de objetos registrados en las listas de *objetos en memoria* y *punteros just-beyond*.

Se observó (con una instancia sin modificar de *htop*) que los procesos *htop* construidos con *IDefBehav* no utilizaban significativamente más procesador que sus variantes nativas sin alteraciones, pero sí que consumían una cantidad notablemente superior de memoria. Esto último se debe a la cantidad de información almacenada por el instrumentador de objetos de *IDefBehav*.

Por último, se obtuvo como resultado que en diferentes tests *htop* incurrió en comportamientos indefinidos capturados exitosamente por la herramienta *IDefBehav*, los cuales se detallan a continuación:

```
chkopsbtwszeroneg
CRT.c:556:98: Operator produced a negative zero in signed magnitude
              representation and implementation may not support it.

chkopsderef, chkopsptraddsub, regdecl
LinuxProcessList.c:637:14: Addition or subtraction produces a
                           result that does not point into, or
                           just beyond, the same array object.

chkopsptraddsub, regdecl
LinuxProcessList.c:637:14: Addition or subtraction produces a
                           result that does not point into, or
                           just beyond, the same array object.

fixopsptrasgn, chkopsptraddsub, regdecl
LinuxProcessList.c:637:14: Addition or subtraction produces a
                           result that does not point into, or
                           just beyond, the same array object.

fixopsptrcmp, chkopsptrcmp, chkopsptraddsub, regdecl
LinuxProcessList.c:637:14: Addition or subtraction produces a
                           result that does not point into, or
                           just beyond, the same array object.
```

de los cuales se pueden identificar únicamente 2 comportamientos indefinidos diferentes.

Una vez reportados estos comportamientos se procedió a la verificación de que los mismos efectivamente fueran comportamientos indefinidos y no errores en el desarrollo de *IDefBehav*. No se abordó ninguna política de corrección de los comportamientos en *htop* ya que sería tema de otro proyecto.

Se puede ver que usando el operador de *shift*, *htop* es capaz de generar un cero negativo en arquitecturas con representación de enteros *signed magnitude*. Sin embargo, viendo el código de *htop* para el reporte del error en `LinuxProcessList.c:637:14` no se puede sospechar que *htop* esté efectivamente haciendo una operación inválida, con lo que queda por averiguar cuál sería un posible *bug* en `idbre` que esté causando este reporte.

Así mismo, en ejecuciones previas de la herramienta *IDefBehav* con *htop*, hubo otros reportes que ayudaron a encontrar errores en la programación de *IDefBehav*, tanto en `idbcc` como en `idbre`, que ocurrían por falta de reportes correctos. Se pudo mejorar así la herramienta.

Un ejemplo de estos casos que se descubrió, fue que se estaban realizando chequeos de comparaciones entre punteros para cualquier operador de comparación, cuando el enunciado del comportamiento menciona ser afectado únicamente por los operadores relacionales y no los de igualdad.

Se encontró también, que si bien *htop* es una utilidad pequeña, a lo largo de una ejecución no muy prolongada deja sin suficiente espacio de registro de objetos a la herramienta *IDefBehav*, delatando la necesidad de una mejor política de liberación de registros por parte de la herramienta.

6 OTRAS HERRAMIENTAS

Durante el desarrollo de este trabajo surgieron en la industria desarrollos que abordan el mismo problema aunque otro enfoque del problema, tanto para *Clang* como para *GCC*.

COMPARACIÓN CON EL UNDEFINEDBEHAVIORSANITIZER DE CLANG

A partir de la versión 3.3 de *Clang* se integró en el desarrollo, del compilador, *UndefinedBehaviorSanitizer* [19], que provino de otro proyecto de *Software Libre*. Su enfoque es un poco diferente, en particular, no se limita a detectar comportamientos indefinidos en el estándar de *C*, sino que también lo hace con el lenguaje *C++*, como resultado de que el proyecto para el que originalmente fue escrita dicha herramienta está desarrollado en este lenguaje.

El *UndefinedBehaviorSanitizer (UBSan)* de *Clang* es un detector de comportamientos indefinidos con la misma idea de *IDefBehav*, de realizar chequeos en tiempo de ejecución de código instrumentado durante la compilación. La principal diferencia entre ambas herramientas radica en el conjunto de chequeadores que cada uno implementa.

En el caso de *UBSan* de *Clang* la lista se encuentra disponible en la página de la herramienta. Para la comparación con los resultados de este trabajo, se decidió correr los tests escritos para *IDefBehav* con los flags `-fsanitize=undefined,integer` y `-fno-sanitize-recover=undefined,integer` de la versión de desarrollo de *Clang* (3.8) al momento de concluir el trabajo. El segundo de los flags empleados es necesario para que *Clang* compile el test de modo que la ejecución finalice al detectar el primer comportamiento indefinido, replicando de este modo el comportamiento de *IDefBehav*.

De los chequeos que *UBSan* realiza, aproximadamente la cuarta parte son únicamente para *C++* que no es parte del alcance de este trabajo, por lo cual no han sido tenidos en cuenta para la comparación.

Tras la ejecución de todos los casos de prueba del modo anteriormente descripto, se pudo observar que *UBSan* detecta una cantidad total de 27 casos de los escritos. Mientras que por su lado, *IDefBehav* es capaz de

detectar 51 casos.

De los 27 casos que *UBSan* detecta, sólo 3 casos no fueron detectados previamente por *IDefBehav*, de los cuales sólo 2 de los tests representaban el comportamiento indefinido deseado y *IDefBehav* no tiene chequeos para ellos; mientras que en el tercero, el test no representaba realmente el comportamiento deseado y en vez se superponía con el comportamiento de otro de los tests que ambas herramientas ya eran capaces de detectar. Además, de los 24 tests restantes, 3 fueron detecciones de los casos de ceros negativos, para los cuales *UBSan* no tiene chequeos. Es decir, los detectó por incurrir en otros comportamientos indefinidos que *IDefBehav* no controla de forma adrede, para estos tests específicamente, por la forma en que fueron construidos los mismos. Es posible lograr el mismo efecto apagando ciertos chequeos de *UBSan*, pero realizar esto no es necesario para una comparación breve.

Se analizaron además algunos de los comportamientos que *IDefBehav* fue capaz de detectar y *UBSan* no, en particular aquellos para los cuales *UBSan* detecta casos similares o sub-casos. Para los casos de chequeos de límites en arreglos, tanto para casos de punteros *just-beyond* como de coordenadas incorrectas en arreglos, *UBSan* falla en la detección de aquellos en los que se apunta a alguna dirección dentro del objeto a detectar con un puntero cualquiera y se opera con este de allí en adelante. En vez, todos aquellos casos en que se utilizó en la operación de punteros la variable que representa el objeto a detectar, *UBSan* realizó, al igual que *IDefBehav*, una detección correcta. Esto probablemente se deba a decisiones de diseño de los creadores de *UBSan* para reducir un impacto desfavorable en el rendimiento del programa instrumentado por esta herramienta.

A modo de resumen, *UBSan* es una herramienta inmediatamente más útil para la industria debido a su rendimiento y a enfocarse en los casos más usuales, sin embargo, considerando el objetivo principal de este trabajo que era cubrir la mayor cantidad de comportamientos posibles, la comparación resulta favorable para *IDefBehav*.

7 CONCLUSIONES

En un principio, el alcance del proyecto se pensó para poder abarcar muchos más casos de los que realmente pudieron ser abordados. Se esperaba poder avanzar de manera más rápida con los casos simples y llegar a casos menos explorados por otras herramientas. Esto no pudo llevarse a cabo debido a que, por las dificultades intrínsecas en el proyecto, se requirió de mucho tiempo de instrumentación de casos medianamente complejos; y de una constante necesidad de revisión de errores por la modificación de partes esenciales del compilador usado como base para la tarea.

La comprensión de los comportamientos indefinidos, como se encuentran redactados en el estándar, resulta difícil pues estos son muchas veces ambiguos o poco claros. De esta manera, la tarea de censar un listado representativo de todos los comportamientos, ya es en sí una tarea que puede introducir numerosos errores y debe ser realizada con mucho cuidado.

La elección de *LLVM* y *Clang* para realizar la tarea fue acertada debido a que gracias a su juventud, tanto *Clang* como *LLVM* gozan de una mayor simpleza en su estructura comparados con otros compiladores con más historia. Además, cuentan con una representación intermedia muy buena, que ayuda con la tarea.

La documentación que provee *LLVM* y *Clang* resulta bastante espartana, lo que dificulta mucho la tarea a la hora de entender el comportamiento de un tipo, o un método. Es de generación automática a partir de los prototipos de las funciones y sus comentarios, y por consiguiente bastante pobre. Además, la documentación oficial no proporciona versiones previas de la misma, con lo que pueden encontrarse diferencias con respecto a la versión utilizada, inesperadamente.

En este trabajo se presentó una herramienta que, sorteando todas las dificultades expuestas, tiene muy buen potencial de crecimiento. Sin esperar que una herramienta como esta pueda ser utilizada a diario, debido a la sobrecarga en el uso de recursos; una vez pulida la implementación, puede resultar ser una herramienta de debugging muy útil y con posibilidades de abarcar incluso los casos que al momento de presentar este trabajo no pudieron ser cubiertos.

8 TRABAJO A FUTURO

Queda como trabajo a futuro portar `idbcc` a la versión actual de *Clang*. Este proyecto produce nuevas versiones aproximadamente dos veces al año. Cuando se comenzó la etapa de implementación de este trabajo, la última versión de *Clang* era la 3.3. Actualmente la última versión es la 3.7.

También queda pendiente implementar la detección de varios de los comportamientos indefinidos seleccionados que no pudo completarse por falta de tiempo.

Muchas de estas implementaciones además dependen de contar con más información en el entorno `idbre` que la recolectada en la versión actual. Se puede implementar la recolección de la información sobre los miembros de uniones y estructuras. También se puede implementar el registro de la entrada y la salida de bloques de código, así como del bloque al que pertenece cada declaración. De esta forma, se puede detectar el uso de objetos fuera de su tiempo de vida.

Muchos comportamientos indefinidos dependen de información que puede variar de una plataforma a otra. La implementación actual resuelve este problema de distintas maneras, en algunos casos sólo detectando el comportamiento indefinido en la plataforma actual, en otros emitiendo un mensaje de error siempre, independientemente de la plataforma actual, si las condiciones se dieran para que el comportamiento indefinido ocurra en alguna. Una posibilidad de trabajo a futuro es, entonces, generalizar la detección de todos los comportamientos indefinidos de este tipo, de modo que la detección sea independiente de la plataforma actual, o que la plataforma para la que se detectan sea configurable.

Continuando con la configuración de la detección, se puede mencionar también el agregar a `idbcc` opciones de línea de comandos para permitir seleccionar la instrumentación que se desea generar. En la implementación actual se pueden *linkear* sólo los módulos del entorno `idbre` deseados, reemplazando los módulos indeseados con *stubs*. Sin embargo, las llamadas a las funciones de `idbre` aún se generan, haciendo más complejo el código *LLVM IR* y produciendo un impacto negativo en el desempeño del programa compilado.

Durante la implementación del entorno `idbre`, la velocidad de ejecución

del programa o su uso de memoria no fue una prioridad. Hay varias oportunidades de lograr mejoras de desempeño utilizando otras estructuras de datos o tipos de datos más compactos. A modo de ejemplo, los tamaños de las dimensiones de los arreglos se almacenan en enteros de 32 bits, pero es muy probable que nunca se necesite un rango de valores tan grande y podría usarse un 75 % menos de memoria para esto utilizando enteros de 8 bits.

Finalmente, queda como trabajo futuro determinar la posibilidad de contribuir alguna parte del código de *IDefBehav* a *Clang* o desarrollarlo hasta el punto de que pueda ser una herramienta útil para la industria.

LISTADO DE COMPORTAMIENTOS RESUELTOS

A continuación se detallan los comportamientos indefinidos abordados por este trabajo y los sub-casos que de ellos se desprenden.

> The arguments to certain operators are such that could produce a negative zero result, but the implementation does not support negative zeros (6.2.6.2).

- Signed magnitude:
  1. Bitwise `and`.
  2. Bitwise `xor`.
  3. Bitwise `or`.
  4. Operador shift izquierdo.
  5. Operador shift derecho.
  6. Bitwise `not`.
- Ones' Complement:
  1. Bitwise `and`.
  2. Bitwise `xor`.
  3. Bitwise `or`.
  4. Operador shift izquierdo.
  5. Operador shift derecho.
  6. Bitwise `not`.

> The operand of the unary * operator has an invalid value (6.5.3.2).

1. Desreferencia de NULL.
2. Desreferencia de punteros mal alineados.

> The value of the second operand of the / or % operator is zero (6.5.5).

1. División entera, con signo.
2. División entera, sin signo.
3. División de punto flotante.
4. Módulo entero, con signo.
5. Módulo entero, sin signo.

Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object (6.5.6).

1. Incremento unario.
2. Decremento unario.
3. Operador de suma.
4. Operador de resta.
5. Cambios en la asociatividad de sub-expresiones de suma y resta.
6. Cambios en la conmutatividad de sub-expresiones de suma y resta.
7. Operador compuesto de suma y asignación (+=).

Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated (6.5.6).

1. Desreferencia de un puntero *just-beyond*.

Pointers that do not point into, or just beyond, the same array object are subtracted (6.5.6).

1. Resta de punteros a objetos diferentes.

An array subscript is out of range, even if an object is apparently accessible with the given subscript (as in the lvalue expression a[1][7] given the declaration int a[4][5]) (6.5.6).

1. Acceso a elementos fuera del arreglo.
2. Acceso a elementos dentro del arreglo usando coordenadas incorrectas.
3. Acceso a elementos fuera del arreglo mediante punteros a subobjetos.
4. Acceso a elementos dentro del arreglo mediante punteros a subobjetos, usando coordenadas incorrectas.

5. Acceso a elementos fuera del arreglo mediante punteros a un elemento del sub-objeto.

6. Acceso a elementos dentro del arreglo mediante punteros a un elemento del sub-objeto usando coordenadas incorrectas.

   The result of subtracting two pointers is not representable in an object of type ptrdiff_t (6.5.6).

1. Underflow de la resta de punteros.

2. Overflow de la resta de punteros.

   An expression is shifted by a negative number or by an amount greater than or equal to the width of the promoted expression (6.5.7).

1. Operador shift izquierdo con desplazamiento negativo.

2. Operador shift derecho con desplazamiento negativo.

3. Operador shift izquierdo con desplazamiento mayor al ancho de bits.

4. Operador shift derecho con desplazamiento mayor al ancho de bits.

   An expression having signed promoted type is left-shifted and either the value of the expression is negative or the result of shifting would be not be representable in the promoted type (6.5.7).

1. Desplazamiento de un valor entero negativo.

2. Desplazamiento de un valor con resulatado no representable en el tipo del valor.

   Pointers that do not point to the same aggregate or union (nor just beyond the same array object) are compared using relational operators (6.5.8).

1. Comparación de punteros a estructuras (`struct`) diferentes.

2. Comparación de punteros a elementos en arreglos diferentes.

3. Comparación de punteros a arreglos diferentes.

   The size expression in an array declaration is not a constant expression and evaluates at program execution time to a non-positive value (6.7.5.2).

1. Declaración de arreglos con tamaños variables menores que cero.

Además, se crearon tests que verifican que las operaciones: aritméticas, relacionales y de asignación, con punteros o direcciones; continúan funcionando correctamente luego de las modificaciones agregadas para manejar punteros *just-beyond*.

B CÓDIGO FUENTE

El código fuente del proyecto, sus programas, bibliotecas y casos de prueba desarrollados para este trabajo se encuentra disponible bajo licencias de *Software Libre* en la *URL*:

`https://gitlab.com/idbhv/idefbehav`

BIBLIOGRAFÍA

[1] *Clang/LLVM Maturity Report*, Moltkestr. 30, 76133 Karlsruhe - Germany, June 2010. *See* http://www.iwi.hs-karlsruhe.de.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] comp.lang.c. C Frequently Asked Questions - Undefined behavior. http://c-faq.com/ansi/undef.html.

[4] D. W. Flater, Y. Yesha, and E.K. Park. Extensions to the C programming language for enhanced fault detection. 23:617–628, 1993.

[5] Free Software Foundation. GCC, the GNU Compiler Collection. http://gcc.gnu.org/.

[6] Richard Jones. A bounds checking C compiler. Technical report, Department of Computing, Imperial College, June 1995.

[7] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG'97. Proceedings of the Third International Workshop on Automatic Debugging*. Linköping University Electronic Press, 1997.

[8] JTC 1/SC 22/WG 14. ISO/IEC 9899:1999: Programming languages – C. Technical report, International Organization for Standards, 1999.

[9] S. C. Kendall. Bcc: run-time checking for C programs. In *Proceedings of the USENIX Summer Conference*, 1983.

[10] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 1988.

[11] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* http://llvm.cs.uiuc.edu.

[12] Chris Lattner. Llvm and clang: Advancing compiler technology. In *Free and Open Source Developers' European Meeting (FOSDEM'11)*, February 2011.

[13] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[14] Hisham Muhammad. htop - an interactive process viewer for Unix. http://hisham.hm/htop/.

[15] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44 – 66, 2003. RV '2003, Run-time Verification (Satellite Workshop of CAV '03).

[16] University of Illinois. clang: a C language family frontend for LLVM. http://clang.llvm.org/.

[17] University of Illinois. Clang Documentation. http://clang.llvm.org/docs/.

[18] University of Illinois. LLVM project home page. http://llvm.org/.

[19] University of Illinois. UndefinedBehaviorSanitizer. http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[20] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[21] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.