

Profesor: Abiel Tomas Parra Hernandez

Materia: Lenguajes y Autómatas 2/ Compiladores

Carrera: Ing. En Sistemas Computacionales.

Proyecto final: "Compiler-rt" runtime libraries

Integrantes del equipo:

- + Andrés Martinez Hernandez.
- + Corina Ruiz Morales.





Introducción

En este proyecto veremos todas las bibliotecas de tiempo de ejecución de un compilador (Compiler-rt- Runtime Library). Se explicará su forma de trabajar y el uso que llegan a tener, a la vez se mostraran algunos ejemplos compilados que nos muestra cuanto tardan algunas bibliotecas en hacer su ejecución y se verán las plataformas compatibles.

El proyecto Compiler-rt consta de:

Builtins: Una biblioteca simple que proporciona una implementación de los ganchos específicos de destino de bajo nivel requeridos por la generación de código y otros componentes de tiempo de ejecución. Por ejemplo, cuando se compila para un destino de 32 bits, convertir un doble en un entero sin signo de 64 bits es compilar en una llamada en tiempo de ejecución a la función "`__fixunsdfdi`". La biblioteca incorporada proporciona implementaciones optimizadas de esta y otras rutinas de bajo nivel, ya sea en forma de C independiente del objetivo o como un ensamblaje altamente optimizado.

builtins proporciona soporte completo para las interfaces `libgcc` en objetivos compatibles e implementaciones de alto rendimiento ajustadas a mano de funciones de uso común como `__floatundidf` en ensamblador que son dramáticamente más rápidas que las implementaciones `libgcc`. Debería ser muy fácil traer incorporados para apoyar a un nuevo objetivo agregando las nuevas rutinas que necesita ese objetivo.

Bibliotecas de tiempo de ejecución que se requieren para ejecutar el código con instrumentación de sanitizer. Esto incluye tiempos de ejecución para:

- ✚ AddressSanitizer.
- ✚ ThreadSanitizer.
- ✚ UndefinedBehaviorSanitizer.
- ✚ MemorySanitizer.
- ✚ LeakSanitizer.
- ✚ DataFlowSanitizer.

AddressSanitizer

AddressSanitizer es un detector de errores de memoria rápido. Consiste en un módulo de instrumentación del compilador y una biblioteca en tiempo de ejecución. La herramienta puede detectar los siguientes tipos de errores:

Accesos fuera de los límites al montón, la pila y los globales

- ✚ Use-after-free
- ✚ Use-after-return (indicador de tiempo de ejecución `ASAN_OPTIONS = detect_stack_use_after_return = 1`)



- ✚ Use-after-scope (bandera de clang `-fsanitize-address-use-after-scope`)
- ✚ Doble libre, inválido gratis
- ✚ Fugas de memoria (experimental)

La ralentización típica introducida por AddressSanitizer es 2x

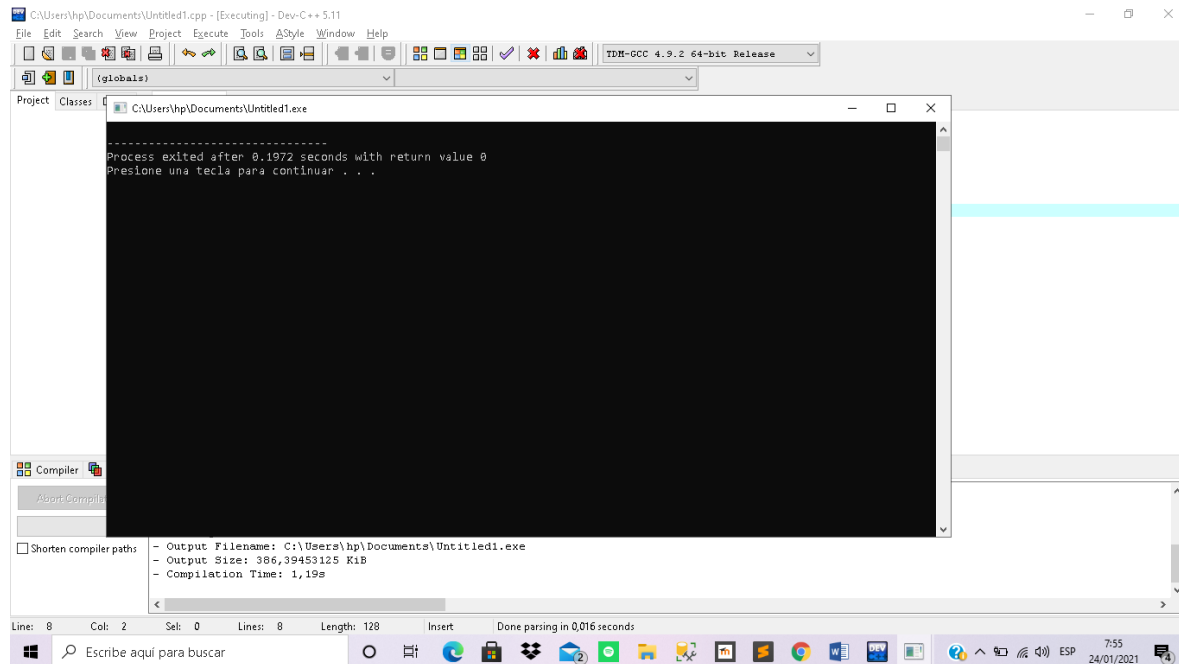
Uso

Simplemente compile y vincule su programa con `-fsanitize=address`. La biblioteca de tiempo de ejecución de AddressSanitizer debe estar vinculada al ejecutable final, así que asegúrese de usar clang(no ld) para el paso de enlace final. Al vincular bibliotecas compartidas, el tiempo de ejecución de AddressSanitizer no está vinculado, por lo que `-Wl,-z,defs` puede causar errores de enlace (no lo use con AddressSanitizer). Para obtener un rendimiento razonable, sume `-O1` superior. Para obtener mejores rastros de pila en los mensajes de error, agregue `-fno-omit-frame-pointer`. Para obtener seguimientos de pila perfectos, es posible que deba deshabilitar la alineación (solo use `-O1`) y la eliminación de llamadas finales (`-fno-optimize-sibling-calls`).

Ejemplo:

```
int main(int argc, char **argv) {  
    int *array = new int[100];  
    delete [] array;  
    return array[argc]; // BOOM  
}
```

Código compilado:



Si se detecta un error, el programa imprimirá un mensaje de error en stderr y saldrá con un código de salida distinto de cero. AddressSanitizer sale con el primer error detectado. Esto es por diseño:

Este enfoque permite a AddressSanitizer producir código generado más rápido y más pequeño (ambos en $\sim 5\%$).

La corrección de errores se vuelve inevitable. AddressSanitizer no produce falsas alarmas. Una vez que se produce un daño en la memoria, el programa se encuentra en un estado incoherente, lo que podría generar resultados confusos e informes posteriores potencialmente engañosos.

Si su proceso es un espacio aislado y está ejecutando OS X 10.10 o anterior, deberá establecer DYLD_INSERT_LIBRARIES la variable de entorno y apuntarla a la biblioteca ASan que está empaquetada con el compilador utilizado para compilar el ejecutable. (Puede encontrar la biblioteca buscando bibliotecas dinámicas con asansu nombre). Si la

variable de entorno no está configurada, el proceso intentará volver a ejecutar. También tenga en cuenta que cuando mueva el ejecutable a otra máquina, la biblioteca ASan también deberá copiarse.

ThreadSanitizer.

ThreadSanitizer es una herramienta que detecta carreras de datos. Consiste en un módulo de instrumentación del compilador y una biblioteca en tiempo de ejecución. La ralentización típica introducida por ThreadSanitizer es de aproximadamente 5x-15x . La sobrecarga de memoria típica introducida por ThreadSanitizer es de aproximadamente 5 x 10 veces.

Plataformas compatibles.

ThreadSanitizer es compatible con los siguientes sistemas operativos:

- + Android aarch64, x86_64
- + Darwin arm64, x86_64
- + FreeBSD
- + Linux aarch64, x86_64, powerpc64, powerpc64le
- + NetBSD

Es posible el soporte para otras arquitecturas de 64 bits, las contribuciones son bienvenidas. El soporte para plataformas de 32 bits es problemático y no está previsto.

Uso

Simplemente compile y vincule su programa con `-fsanitize=thread`. Para obtener un rendimiento razonable, sume `-O1` superior. Úselo `-g` para obtener nombres de archivo y números de línea en los mensajes de advertencia.

Ejemplo:

```
% cat projects/compiler-rt/lib/tsan/lit_tests/tiny_race.c

#include <pthread.h>

int Global;

void *Thread1(void *x) {
    Global = 42;
    return x;}

int main() {
    pthread_t t;

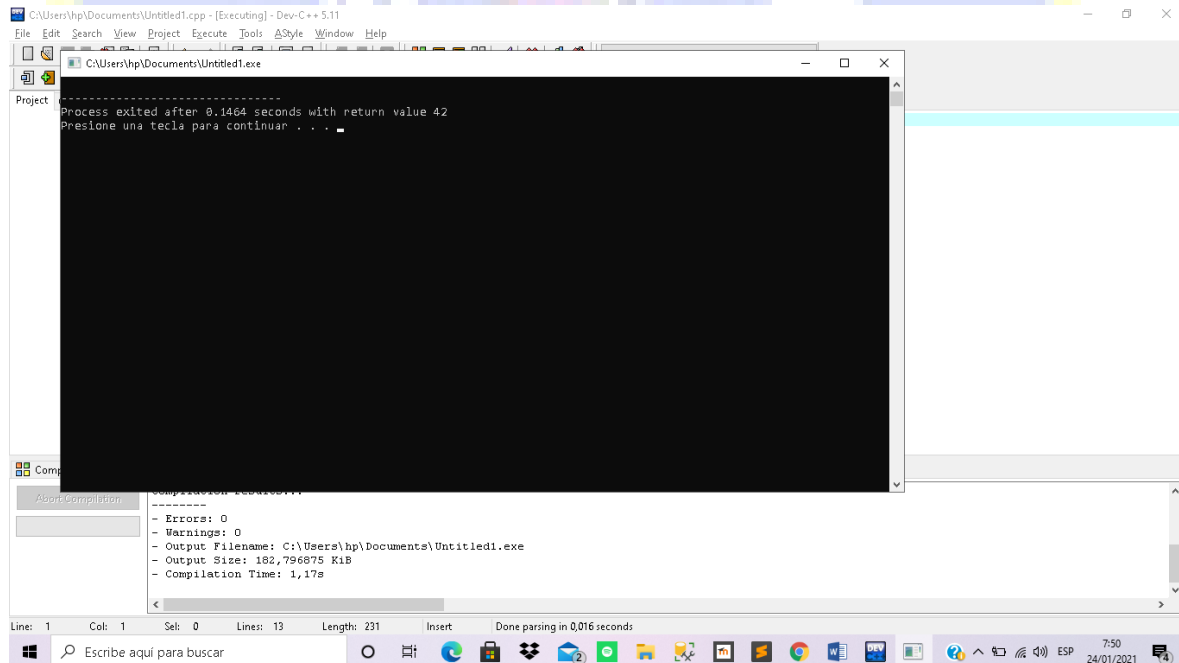
    pthread_create(&t, NULL, Thread1, NULL);

    Global = 43;

    pthread_join(t, NULL);

    return Global;}
```

Código compilado:



```
C:\Users\hjp\Documents\Untitled1.exe
Process exited after 0.1464 seconds with return value 42
Presione una tecla para continuar . . .
```

- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\hjp\Documents\Untitled1.exe
- Output Size: 182,796875 Kib
- Compilation Time: 1,17s

Si se detecta un error, el programa imprimirá un mensaje de error en stderr. Actualmente, ThreadSanitizer simboliza su salida mediante un addr2lineproceso externo (esto se solucionará en el futuro).



UndefinedBehaviorSanitizer.

El compilador Clang es un compilador de código abierto para la familia de lenguajes de programación C, con el objetivo de ser la mejor implementación de estos lenguajes en su clase. Clang se basa en el generador de código y el optimizador LLVM, lo que le permite proporcionar optimización de alta calidad y compatibilidad con la generación de código para muchos objetivos. Para obtener más información general, consulte el sitio web de Clang o el sitio web de LLVM .

Este documento describe notas importantes sobre el uso de Clang como compilador para un usuario final, la documentación de las funciones admitidas, las opciones de la línea de comandos, etc. Si está interesado en usar Clang para crear una herramienta que procese código, consulte "Clang" CFE Internals Manual . Si está interesado en el analizador estático de Clang , consulte su página web.

Clang es un componente en una cadena de herramientas completa para los lenguajes de la familia C. Un documento separado describe las otras piezas necesarias para ensamblar una cadena de herramientas completa.

Clang está diseñado para admitir la familia C de lenguajes de programación, que incluye C , Objective-C , C ++ y Objective-C ++ , así como muchos dialectos de esos. Para obtener información específica del idioma, consulte la sección específica del idioma correspondiente:

- ✚ Lenguaje C : K&R C, ANSI C89, ISO C90, ISO C94 (C89 + AMD1), ISO C99 (+ TC1, TC2, TC3).
- ✚ Lenguaje Objective-C : ObjC 1, ObjC 2, ObjC 2.1, más variantes según el idioma base.
- ✚ Lenguaje C ++
- ✚ Lenguaje objetivo C ++
- ✚ Lenguaje del kernel de OpenCL : OpenCL C v1.0, v1.1, v1.2, v2.0, más C ++ para OpenCL.

Además de estos idiomas base y sus dialectos, Clang admite una amplia variedad de extensiones de idioma, que están documentadas en la sección de idioma correspondiente. Estas extensiones se proporcionan para ser compatibles con GCC, Microsoft y otros compiladores populares, así como



para mejorar la funcionalidad a través de características específicas de Clang. El controlador Clang y las características del lenguaje están diseñadas intencionalmente para ser tan compatibles con el compilador GNU GCC como sea razonablemente posible, facilitando la migración de GCC a Clang. En la mayoría de los casos, el código "simplemente funciona". Clang también proporciona un controlador alternativo, clang-cl, que está diseñado para ser compatible con el compilador de Visual C++, cl.exe.

Además de las características específicas del idioma, Clang tiene una variedad de características que dependen de la arquitectura de la CPU o del sistema operativo para el que se compila. Consulte la sección Limitaciones y características específicas de Target para obtener más detalles.

El resto de la introducción presenta una terminología básica de compiladores que se utiliza en este manual y contiene una introducción básica al uso de Clang como compilador de línea de comandos.

Terminología

Interfaz, analizador, backend, preprocesador, comportamiento indefinido, diagnóstico, optimizado.

Uso básico

Introducción a cómo usar un compilador de C para principiantes.

Compilar + vincular compilar luego vincular información de depuración que permite optimizaciones eligiendo un idioma para usar, por defecto es C17. Detecciones automáticas basadas en extensión. usando un archivo MAKE

Opciones de línea de comando

Esta sección es generalmente un índice de otras secciones. No profundiza en los que se tratan en otras secciones. Sin embargo, la primera parte presenta la selección de idioma y otras opciones de alto nivel como -c, -g, etc.

MemorySanitizer.

MemorySanitizer es un detector de lecturas no inicializadas. Consiste en un módulo de instrumentación del compilador y una biblioteca en tiempo de ejecución.

La ralentización típica introducida por MemorySanitizer es 3x.

Cómo construir

Compile LLVM / Clang con CMake.

Uso

Simplemente compile y vincule su programa con -fsanitize=memoryflag. La biblioteca de tiempo de ejecución MemorySanitizer debe estar vinculada al ejecutable final, así que asegúrese de usar clang(no ld) para el paso de vínculo final. Al vincular bibliotecas compartidas, el tiempo de ejecución de MemorySanitizer no está vinculado, por lo que -Wl,-z,defspuede causar errores de enlace (no lo use con MemorySanitizer). Para obtener un rendimiento razonable, sume -O1o superior. Para obtener seguimientos de pila significativos en los mensajes de error, agregue -fno-omit-frame-pointer. Para obtener seguimientos de pila perfectos, es posible que deba deshabilitar la alineación (solo use -O1) y la eliminación de llamadas finales (-fno-optimize-sibling-calls).

Ejemplo:

```
#include <stdio.h>

int main(int argc, char** argv) {

    int* a = new int[10];

    a[5] = 0;

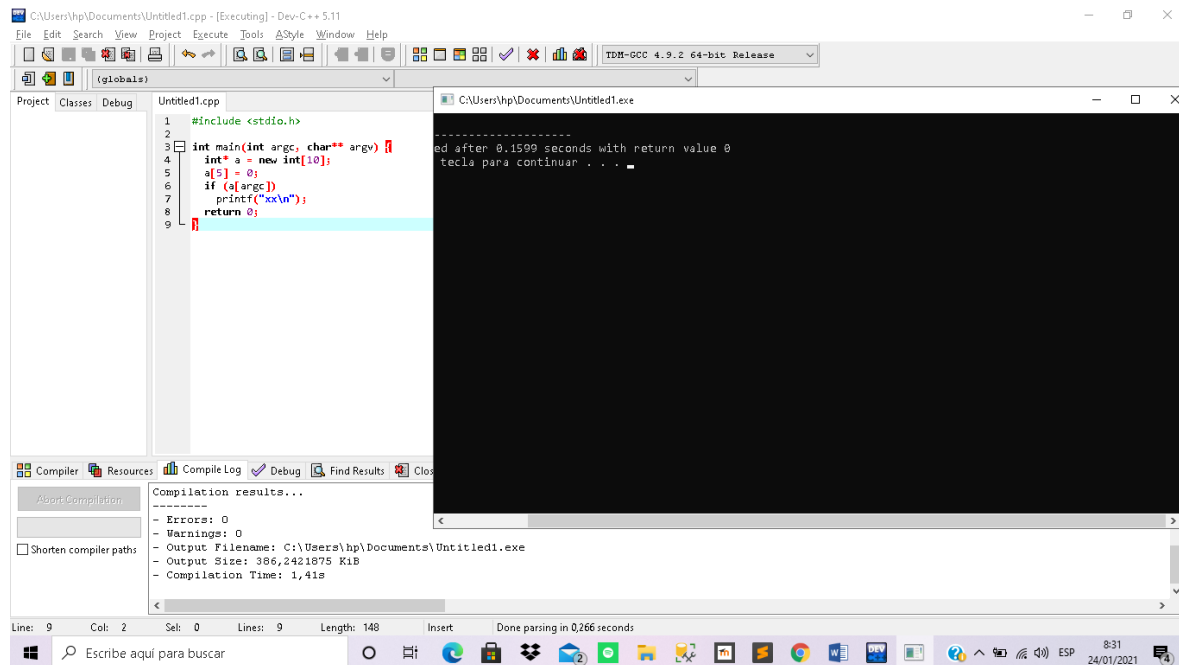
    if (a[argc])

        printf("xx\n");

    return 0;

}
```

Código compilado:



```
#include <stdio.h>

int main(int argc, char** argv) {
    int* a = new int[10];
    a[5] = 0;
    if (a[argc])
        printf("xx\\n");
    return 0;
}
```

Compilation results...

- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\hp\Documents\Untitled1.exe
- Output Size: 386,2421875 KiB
- Compilation Time: 1,41s

Si se detecta un error, el programa imprimirá un mensaje de error en stderr y saldrá con un código de salida distinto de cero.

LeakSanitizer.

LeakSanitizer es un detector de fugas de memoria en tiempo de ejecución. Puede combinarse con AddressSanitizer para obtener tanto errores de memoria como detección de fugas, o usarse en modo independiente. LSan casi no agrega gastos generales de rendimiento hasta el final del proceso, momento en el que hay una fase adicional de detección de fugas.

Uso

LeakSanitizer es compatible con x86_64 Linux y macOS. Para usarlo, simplemente cree su programa con AddressSanitizer:

Ejemplo:

```
#include <stdlib.h>

void *p;

int main() {

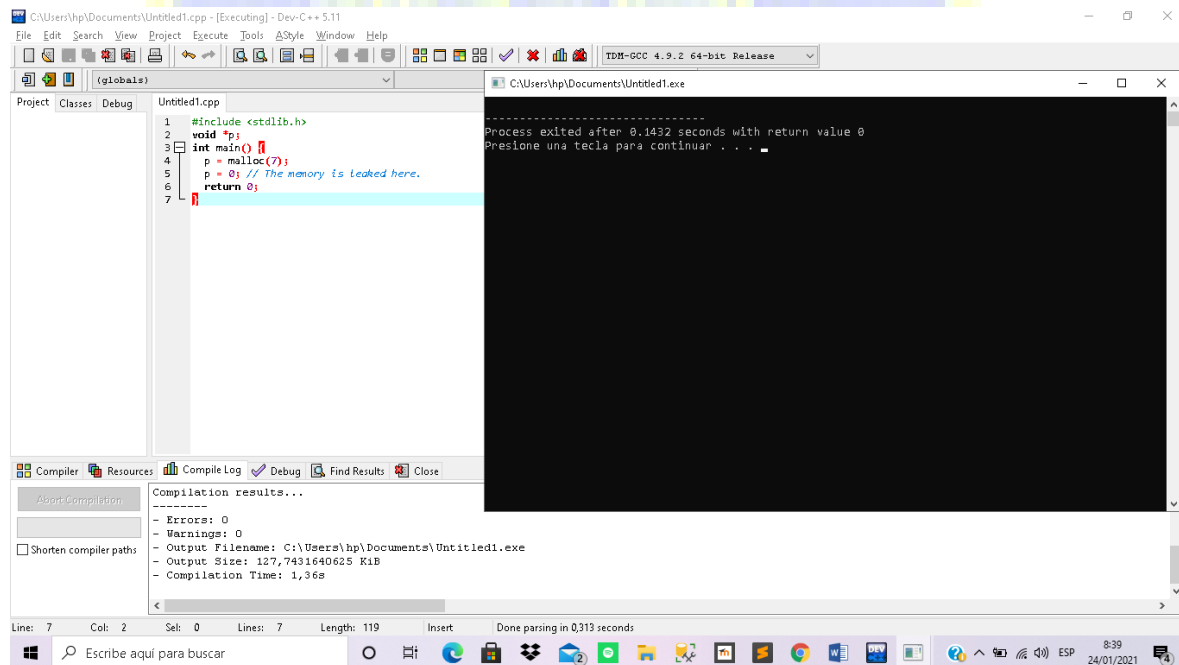
    p = malloc(7);

    p = 0; // The memory is leaked here.

    return 0;

}
```

Código compilado



Para usar LeakSanitizer en modo independiente, vincule su programa con `-fsanitize=leakflag`. Asegúrese de usar clang(no ld) para el paso de enlace, de modo que se vincule en la biblioteca de tiempo de ejecución de LeakSanitizer adecuada al ejecutable final.

DataFlowSanitizer.

DataFlowSanitizer es un análisis de flujo de datos dinámico generalizado.

A diferencia de otras herramientas Sanitizer, esta herramienta no está diseñada para detectar una clase específica de errores por sí sola. En cambio, proporciona un marco genérico de análisis de flujo de datos dinámicos que los clientes pueden utilizar para ayudar a detectar problemas específicos de la aplicación dentro de su propio código.

Cómo construir libc ++ con DFSan

DFSan requiere que todo su código esté instrumentado o que las funciones no instrumentadas se enumeren como uninstrumented en la lista ABI.

Si desea tener funciones libc ++ instrumentadas, debe compilarlas con la instrumentación DFSan de la fuente. A continuación, se muestra un ejemplo de cómo compilar libc ++ y la ABI de libc ++ con instrumentación de desinfección de flujo de datos.

Uso

Sin cambios en el programa, la aplicación de DataFlowSanitizer a un programa no alterará su comportamiento. Para usar DataFlowSanitizer, el programa usa funciones API para aplicar etiquetas a los datos para hacer que se rastreen y para verificar la etiqueta de un elemento de datos específico. DataFlowSanitizer gestiona la propagación de etiquetas a través del programa de acuerdo con su flujo de datos.

Las API se definen en el archivo de encabezado sanitizer/dfsan_interface.h. Para obtener más información sobre cada función, consulte el archivo de encabezado.

Lista ABI

DataFlowSanitizer usa una lista de funciones conocida como lista ABI para decidir si una llamada a una función específica debe usar la ABI nativa del sistema operativo o si debe usar una variante de esta ABI que también propaga etiquetas a través de parámetros de función y valores de retorno.



El archivo de lista ABI también controla cómo se propagan las etiquetas en el primer caso. DataFlowSanitizer viene con una lista ABI predeterminada que está destinada a cubrir eventualmente la biblioteca glibc en Linux, pero puede ser necesario que los usuarios extiendan la lista ABI en los casos en que una biblioteca o función en particular no se pueda instrumentar (por ejemplo, porque está implementada en ensamblador o otro idioma que DataFlowSanitizer no admite) o se llama a una función desde una biblioteca o función que no se puede instrumentar.

El archivo de lista ABI de DataFlowSanitizer es una lista de casos especiales de Sanitizer. El pase trata todas las funciones de la uninstrumentedcategoría en el archivo de lista ABI como conformes con la ABI nativa. A menos que la lista ABI contenga categorías adicionales para esas funciones, una llamada a una de esas funciones producirá un mensaje de advertencia, ya que se desconoce el comportamiento de etiquetado de la función. Las otras categorías son compatibles discard, funcional y custom.

- ✚ discard- En la medida en que esta función escribe en la memoria (accesible para el usuario), también actualiza las etiquetas en la memoria secundaria (esta condición se cumple trivialmente para las funciones que no escriben en la memoria accesible para el usuario). Su valor de retorno no está etiquetado.
- ✚ functional- Me gusta discard, excepto que la etiqueta de su valor de retorno es la unión de la etiqueta de sus argumentos.
- ✚ custom- En lugar de llamar a la función, `__dfsw_F` se llama a un contenedor personalizado, donde `F` es el nombre de la función. Esta función puede envolver la función original o proporcionar su propia implementación. Esta categoría se usa generalmente para funciones no instrumentables que escriben en la memoria accesible para el usuario o que tienen un comportamiento de propagación de etiquetas más complejo. La firma de `__dfsw_F` se basa en la de `F` y cada argumento tiene una etiqueta de tipo `dfsan_label` adjunta a la lista de argumentos. Si `F` es de tipo de retorno no nulo, se agrega un argumento final de tipo `al` que la función personalizada puede almacenar la etiqueta para el valor de retorno.



Clientela

Actualmente, los proyectos Clang y LLVM utilizan principalmente compiler-rt como implementación para las bibliotecas de soporte del compilador en tiempo de ejecución. Para obtener más información sobre el uso de compiler-rt con Clang, consulte la página de introducción a Clang.

Soporte de plataforma

Se sabe que los builtins funcionan en las siguientes plataformas:

Arquitecturas de máquinas: i386, X86-64, SPARC64, ARM, PowerPC, PowerPC 64.

SO: AuroraUX, DragonFlyBSD, FreeBSD, NetBSD, Linux, Darwin.

La mayoría de los tiempos de ejecución de desinfectantes solo son compatibles con Linux x86-64. Consulte las páginas específicas de la herramienta en los documentos de Clang para obtener más detalles.

Estructura de la fuente

Una breve explicación de la estructura de directorios de Compiler-rt:

Para las pruebas, es posible construir una biblioteca genérica y una biblioteca optimizada. La biblioteca optimizada se forma superponiendo las versiones optimizadas en la biblioteca genérica. Por supuesto, algunas arquitecturas tienen funciones adicionales, por lo que la biblioteca optimizada puede tener funciones que no se encuentran en la versión genérica.

- ✚ incluir / contiene encabezados que se pueden incluir en los programas de usuario (por ejemplo, los usuarios pueden llamar directamente a determinadas funciones desde los tiempos de ejecución del desinfectante).
- ✚ lib / contiene implementaciones de bibliotecas.
- ✚ lib / builtins es una implementación portátil genérica de rutinas integradas .
- ✚ lib / builtins / (arch) tiene versiones optimizadas de algunas rutinas para las arquitecturas compatibles.
- ✚ test / contiene conjuntos de pruebas para tiempos de ejecución de compilador-rt.