Klee “Symbolic virtual machine”

Instituto tecnológico de Iztapalapa

Lenguajes y autómatas II

Profesor: Parra Hernández Abiel Tomas

Grupo: ISC – 7AV / SCD1016

Nombre del equipo: Yami

Integrantes del equipo: No. Ctrl:

Guerrero Verde Geovanni 161080010
Rojas Hernández Axel Joel 181080176
Sánchez Morales Luis Daniel 161080019
Oscar Lucero Zavaleta 171080170

Introducción

En este proyecto de investigación sobre KLEE Symbolic Execution Engine está plasmado el trabajo realizado de varias semanas por lo cual se encontrarán con información detalla de la herramienta de código abierto, así como ejemplos realizados en ella. El trabajo realizado por varios compañeros en este artículo de información es que conozcan lo que es KLEE la herramienta de código abierto desarrollada por Daniel Dunbar el autor principal de esta gran herramienta de ejecución simbólica dinámica que fue desarrollada en una de las grandes universidades del mundo, la Universidad de Stanford y ahora desarrollada y mantenida principalmente por Software Reliability Group en Imperial College London. KLEE tiene una gran comunidad que abarca tanto la academia como la industria, con más de 60 colaboradores en GitHub, más de 350 suscriptores en su lista de correo y más de 80 participantes en un taller dedicado reciente. KLEE ha sido utilizado y extendido por grupos de muchas universidades y empresas en una variedad de áreas diferentes, como generación de pruebas de alta cobertura, depuración automatizada, generación de exploits, redes de sensores inalámbricos y juegos en línea, entre muchas otras.

Lo que hace más genial a KLEE, es que al modelar rutas utilizando fórmulas matemáticas, tiene aplicaciones para una amplia variedad de problemas,más allá de la generación de pruebas y la búsqueda de errores.

Ejemplos:

- incluyendepuración
- inferencia de especificación
- reparación de entradas y programas
- reproducción de fallas
- análisis de parches.

En todo caso KLEE ha mejorado la experiencia de mis compañeros y mía al conocer esta gran herramienta de ejecución simbólica dinámica, ya que para los que estamos en la rama de la computación o para los que les gusta aprender cosas sobre la materia con esta herramienta se sorprenderán porqueproporciona la capacidad de explorar automáticamente las rutas en un programa, utilizando un solucionador de restricciones para razonar sobre la viabilidad de la ruta, aparte que tiene ejemplos en su página con los cual puedes empezar aprende sobre esta gran herramienta.

Justificación

Con la realización del presente proyecto de investigación se logrará conocer sobre la herramienta de código abierto KLEE Symbolic Execution Engine se plasman los conocimientos teóricos adquiridos en clasey en la página webde KLEE [1]

La intención de este este proyecto es conocer más a fondo sobre KLEE y como ayuda este a la realización de problemas que incluyan “operaciones matemáticas” avanzadas, generación de pruebas,búsqueda de errores y no solo esotiene aplicaciones para una amplia variedad de problemas.

Todo esto se llevó a la práctica realizando algunos ejemplos que se encontrar más adelante en la investigación. La experiencia adquirida en este proyecto de investigación que nos llevamos tanto mis compañeros y yo es de 50% teórico y 50% practico, de esa manera nos preparamos para el mundo laboral que nos espera, ya que con la práctica se lograra tener los conocimientos necesarios para un buen desempeño en nuestra rama y ser competitivos en todo lo que hagamos.

La importancia de este proyecto no es más que dar a conocer lo que es esta herramienta de ejecución simbólica dinámicay para qué sirve, lo que mi profesor y compañeros queremos es que se interesen por esta gran herramienta y puedan hacer grandes cosas. Esto fue de gran beneficio para mis compañeros y yo porque aprendíamos de esta gran herramienta y cualquier persona que esté en la rama de informática puede aprender sin ningún problema. Hemos puesto empeño en este proyecto para adquieran todo conocimiento sobre la herramienta KLEE y sobre todo brindar de nuestro conocimiento a quien más lo necesite.

Contenido

Klee – Motor de ejecución simbólica 5
Ejecución simbólica 5
¿Qué son las pruebas de caja blanca? 7
¿Que son las pruebas de caja negra? 8
Entorno simbólico 8
Entorno 8
Simbólico 8
Ejecución simbólica con Klee 11
Prueba del primer tutorial de Klee – Una pequeña función 13
Instalar en pc 13
Configurar un proyecto de Windows para usar herramientas Clang 14
Configurar un proyecto de Linux para usar herramientas Clang 14
Establecer una ubicación LLVM personalizada 15
Establezca propiedades adicionales, edite, compile y depure 16
Prueba del primer tutorial de Klee – Una pequeña función 16
Marcar la entrada como simbólica 17
Compilación en código de bits LLVM 17
Ejecutando KLEE 18
Casos de prueba generados por KLEE 18
Reproducción de un caso de prueba 20
Prueba del segundo tutorial – Resolviendo un laberinto 21
¿Cómo marcar la parte del código que nos interesa? 26
Prueba de programas Rust con klee 29
Bibliografía 34

Klee – Motor de ejecución simbólica

Klee es un motor de ejecución simbólica dinámica construido sobre la infraestructura del compilador LLVM y disponible bajo la licencia de código abierto VIVIC.

Antes de entrar en explicaciones sobre el funcionamiento de KLEE debemos entender que es una "ejecución simbólica".

Ejecución simbólica

La evaluación y ejecución simbólica es un análisis donde el análisis se define como una inspección para entender las propiedades y capacidades del objeto en cuestión.

Con la evaluación y ejecución tratamos de explotar la ventaja del razonamiento general.

Ejemplo

```
read (a);  read (b);
x=a+1;  y=x*b;
print(y)(
```

} El resultado siempre es: (a+1)*b

El razonamiento, al ser más general, me permite evaluar más propiedades que el cálculo general no lo permite.

$$Sea\ f(x)\ una\ función\ que\ queremos\ testear:$$

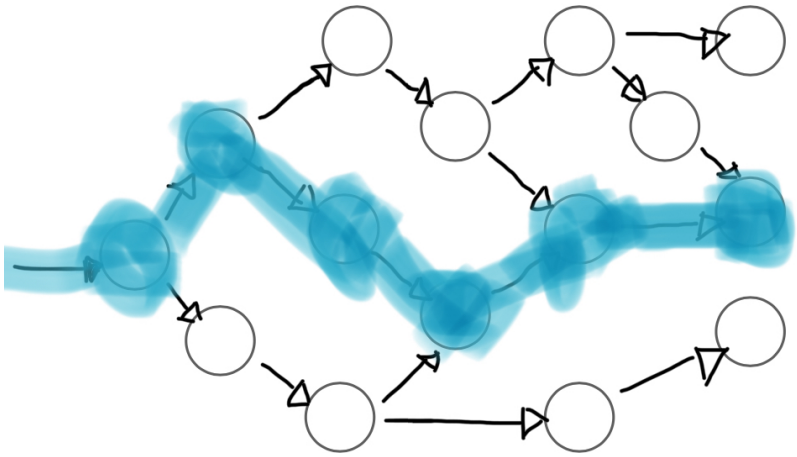
$$f(n) == (n * 2)^3 \cdots ¿?$$

Probamos para un símbolo "n", si el resultado de ejecución simbólica es lo esperado, entonces funciona para cualquier n.

La ejecución simbólica trata de hacer las cosas más generales.

La evaluación simbólica nos permite determinar las condiciones que deben ser verificadas por los datos de entrada para que un camino en particular se ejecute.

Motor de ejecución simbólica



"Para poder hacer testing se necesitan datos"

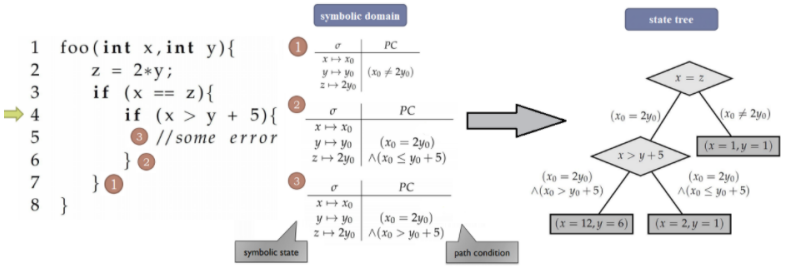
También nos sirve para determinar la relación entre los valores ingresados por la ejecución del programa.

Si yo quiero recorrer un camino en específico, necesito poder descubrir que características deben que tener los datos de entrada para forzar dicho camino.

Comentado [Z21]: Al final de algunos textos habrá textos con fondo color turquesa, son conclusiones personales de algunos temas.

La ejecución simbólica es una técnica de análisis, una técnica de análisis es la agrupación de los distintos elementos individuales que forman el todo (cuenta o partida determinada) de tal manera, que los grupos conformados constituyan unidades homogéneas de estudio. Consiste en ir de lo general a lo específico (método deductivo) con el propósito de examinar con responsabilidad y bajo el criterio de razonabilidad el que las operaciones se ajusten a la Ley, los estatutos, procedimientos, políticas y manuales de la compañía.

En la técnica del Análisis se descompone el sistema en elementos de más fácil manejo, para su estudio y posterior recomposición o síntesis (inducción), sin olvidar que estas partes así estudiadas continúan formando parte del todo, por lo cual no pueden omitirse sus relaciones.



¿Cómo lo hacemos?

PC (Path condition)

El PC representa la condición booleana de ejecución en su camino, Su valor, es una expresión simbólica booleana.

La condición de ejecución de un camino permite saber si el camino es ejecutable. Permite derivar los datos que causan la ejecución de dicho camino.

```
1.  PROGRAM ATP
2.  VAR
3.    x, p : INTEGER;
4.  BEGIN
5.    read(x);
6.    read(p);
7.    p = ABS(p);
8.    x = p;
9.    IF (x >= 0)
10.     THEN write(x);
11.     ELSE write(0);
12.   END
```

```
1.  { (x = undef), (x = undef), (PC = true) }
2.
3.
4.
5.  { (x = Q), (p = undef), (PC = true) }
6.  { (x = Q), (p = A), (PC = true) }
7.  { (x = Q), (p = |A|), (PC = true) }
8.  { (x = |A|), (p = |A|), (PC = true) }
9.  [(x >= 0)] -> [(|A| >= 0)] -> Siempre true
```

¿Qué son las pruebas de caja blanca?

Las pruebas de caja blanca se centran en los detalles procedimentales del software, por lo que su diseño está fuertemente ligado al código fuente.

El ingeniero de pruebas escoge distintos valores de entrada para examinar cada uno de los posibles flujos de ejecución del programa y cerciorarse de que se devuelven los valores de salida esperados.

Aunque las pruebas de caja blanca son aplicables a varios niveles, habitualmente se aplican a las unidades de software. Su cometido es comprobar los flujos de ejecución dentro de cada unidad, pero también pueden probar los flujos entre unidades durante la integración e incluso entre subsistemas, durante las pruebas de sistema.

A pesar de que este enfoque permite diseñar pruebas que cumplan una amplia variedad de casos de prueba, podría pasar por alto partes incompletas de la especificación o requisitos faltantes, pese a garantizar la prueba exhaustiva de todos los flujos de ejecución del código analizado. Las principales técnicas son:

- Pruebas de flujo de control
- Pruebas de flujo de datos
- Pruebas de bifurcación (Branch Testing)
- Pruebas de caminos básicos

Su uso en hacking

En los análisis de vulnerabilidades y pruebas de penetración de sistemas informáticos las pruebas de caja blanca hacen referencia a una metodología donde el ciberdelincuente posee conocimiento total y absoluto del sistema que pretende atacar.

El objetivo de estos test, que perciben el sistema de forma transparente, es similar el comportamiento de un intruso malicioso que contase con permisos de acceso e información precisa acerca del sistema.

¿Que son las pruebas de caja negra?

En teoría de sistemas y física, una caja negra es un elemento que se estudia desde el punto de vista de las entradas que recibe y las salidas o respuestas que produce, sin tener en cuenta su funcionamiento interno. En otras palabras, de una caja negra nos interesará su forma de interactuar con el medio que le rodea (en ocasiones, otros elementos que también podrían ser cajas negras) entendiendo qué es lo que hace, pero sin dar importancia a cómo lo hace. Por tanto, de una caja negra deben estar muy bien definidas sus entradas y salidas, es decir, su interfaz; en cambio, no se precisa definir ni conocer los detalles internos de su funcionamiento.

En programación modular, donde un programa (o un algoritmo) es dividido en módulos, en la fase de diseño se buscará que cada módulo sea una caja negra dentro del sistema global que es el programa que se pretende desarrollar, de esta manera se consigue una independencia entre los módulos que facilita su implementación separada por un equipo de trabajo donde cada miembro va a encargarse de implementar una parte (un módulo) del programa global; el implementador de un módulo concreto deberá conocer como es la comunicación con los otros módulos (la interfaz), pero no necesitará conocer cómo trabajan esos otros módulos internamente; en otras palabras, para el desarrollador de un módulo, idealmente, el resto de módulos serán cajas negras. Es muy importante.

Entorno simbólico

Antes de explicar el concepto de un entorno simbólico, debemos entender el significado de entorno y simbólico.

Entorno

Según la rae, el entorno es el ambiente o lo que nos rodea.

En informática es el conjunto de características que definen el lugar y la forma de ejecución de una aplicación.

Simbólico

Las personas nos comunicamos a través de una herramienta muy eficaz, el lenguaje. Al mismo tiempo, los gestos también expresan ideas o actitudes. Sin embargo, hay otro elemento

que también forma parte de la comunicación: los símbolos, Cualquier cosa tiene un doble valor, por lo que es propiamente y aquello que representa, su simbolismo.

El valor simbólico de un objeto es su capacidad para transmitir unas ideas. La cruz cristiana es un elemento reconocido en todo el mundo y se utiliza en múltiples lugares (en el hogar, las iglesias o los cementerios). En cualquier caso, la cruz significa mucho más que dos palos cruzados. Lo mismo sucede con una gran cantidad de objetos. El símbolo es la representación de una cosa, una idea o una entidad. Por ejemplo, el escudo de un equipo de fútbol explica unos sentimientos y una tradición.
Lo simbólico interviene en la comunicación como si fuera una síntesis de un mensaje complejo. El objeto del símbolo es un resumen de una historia. Es lo que sucede con los símbolos nacionales: la bandera, el traje tradicional, una bebida o un personaje muy importante en un país. Cualquier elemento, por insignificante que sea, tiene un potencial valor simbólico. Y cuando lo vemos, automáticamente pensamos en su significado. Gracias a los símbolos somos capaces de entender rápidamente nuestro entorno. Las marcas comerciales son cuidadosas a la hora de elaborar sus logos, ya que éstos tienen un impacto visual muy potente y son un mecanismo eficaz en la estrategia comercial de la entidad.
Por otro lado, lo simbólico no sólo se refiere a aquello que es compartido por un colectivo, sino que cada individuo otorga un sentido personal a algunas cosas que le rodean. Un libro, un reloj, un anillo o un mechón de pelo tienen un simbolismo para quien relaciona el objeto con un recuerdo especial.

Cada grupo humano comparte unas ideas y tradiciones y al mismo tiempo va creando su simbología. Es una manera de compartir algo, de establecer lazos en una comunidad.
De forma inconsciente, cuando visitamos un sitio desconocido estamos atentos al ambiente general y necesitamos entender el significado del nuevo panorama. Para hacerlo, observamos cuáles son los símbolos que aparecen, qué sentido tienen y cómo se usan. Si no hiciéramos este ejercicio intelectual sería imposible entender la mentalidad del lugar en el que estamos.

Motor de ejecución simbólica

En las Tic hay muchas formas de entorno simbólico y estas presentan potencial para el aprendizaje y la conectividad para el trabajo.

- Formalismo: Se refiere a la serie de instrucciones secuenciales definidas, rígidas que nos vemos obligados a seguir, en ocasiones, cuando hacemos uso de las herramientas TIC.
- Implica previsión y planificación de las acciones.
- Favorece la toma de conciencia y la autorregulación.
- Interactividad: Se refiere a la capacidad que tienen las TIC para el usuario establezca una relación inmediata entre la información y sus propias acciones de búsqueda o procesamiento de esta.
- Permite una relación más activa y consistente con la información.
- Potencia el protagonismo del aprendiz.
- Facilita la adaptación a distintos ritmos de aprendizaje.
- Tiene efectos positivos para la motivación y la autoestima.
- Dinamismo: Los entornos TIC permiten transmitir informaciones dinámicas, es decir, informaciones que evolucionan y se transforman a medida que se van representando y transmitiendo.
- Ayuda a trabajar con simulaciones de situaciones reales.

- Permite interactuar con realidades virtuales.
- Favorece la exploración y la experimentación.

- Multimedia: Las TIC permiten combinar tecnologías específicas y los sistemas y formatos de representación propios de cada uno de ellos.

- Permite la integración, la complementariedad y el tránsito entre diferentes sistemas y formatos de representación.
- Facilita la generalización del aprendizaje.

- Hipermedia: Las TIC ofrecen la posibilidad de navegar entre informaciones que utilizan diferentes “media”

- Comporta la posibilidad de establecer formas diversas y flexibles de organización de las informaciones, estableciendo relaciones múltiples y diversas entre ellos.
- Facilita la autonomía, la exploración y la indagación.
- Potencia el protagonismo en el aprendiz.

- Conectividad: Las posibilidades que ofrecen los entornos basados en las TIC para establecer redes de información y comunicación con múltiples puntos de acceso.

- Permite el trabajo en red de agentes educativos y aprendices. Abre nuevas posibilidades al trabajo grupal y colaborativo.
- Facilita la diversificación en cantidad y calidad, de las ayudas que los agentes educativos ofrecen a los aprendices.

Por lo tanto, un motor de ejecución simbólica dinámica seria:
Respuesta pendiente

Ejecución simbólica con Klee

KLEE es un motor de ejecución simbólica que se puede utilizar para automatizar la generación de casos de prueba, así como para encontrar errores. Es de código abierto y se origina en el mundo académico, y se ha mantenido y desarrollado activamente durante más de una década.

Video 1: Introducción a la ejecución simbólica con KLEE

El primer video muestra cómo instalar la imagen de Docker que viene con el código fuente de KLEE y cómo ejecutar KLEE desde dentro de la imagen de Docker. Luego, el video pasa por los pasos necesarios para ejecutar una aplicación C simple en el entorno KLEE, es decir, cómo usar clangpara extraer el código de bits LLVM del archivo dado y luego usar KLEE para analizar el código LLVM resultante. El video muestra cómo usar la klee_make_symbolicfunción para simbolizar la memoria, cómo KLEE genera múltiples casos de prueba que exploran diferentes rutas de ejecución en el programa y cómo podemos usar KLEE.ktest-toolinspeccionar los valores concretos de los datos simbólicos. Luego, el video pasa por varios de los ejemplos del sitio web de KLEE y muestra lo que sucede cuando KLEE detecta un error en el programa de destino y cómo hacer un análisis de causa raíz basado en la salida de KLEE.

Un usuario puede comenzar a verificar muchos programas reales con KLEEen segundos: KLEE normalmente no requiere modificaciones de fuente ni trabajo manual. Los usuarios primero compilan su códigoa bytecode usando el compilador LLVM disponible públicamente para GNU C. Compilamos tr usando:llvm-gcc --emit-llvm -c tr.c -o tr.bcLos usuarios luego ejecutan KLEE en el bytecode generado, indicando opcionalmente el número, tamaño y tipo de entradas simbólicaspara probar el código. Para tr usamos el comando:klee --max-time 2 --sym-args 1 10 10--sym-files 2 2000 --max-fail 1 tr.bcLa primera opción, --max-time, le dice a KLEE que verifiquetr.bc durante dos minutos como máximo. El resto describe las entradas simbólicas. La opción --sym-args 1 10 10dice que use de cero a tres argumentos de línea de comando, elprimero 1 carácter de largo, los otros 10 caracteres de largo. La opción --sym-files 2 2000 dice usar estándarentrada y un archivo, cada uno con 2000 bytes de simbólicosdatos. La opción --max-fail 1 dice fallar como máximouna llamada al sistema a lo largo de cada ruta de programa.

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                             9
3 :   while (*arg) {                                         10*
4 :     if (*arg == '\\') {                                  11*
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:           i++;
11:         } while (i<4 && *arg>='0' && *arg<='7');
12:         *buffer++ = ac;
13:       } else if (*arg != '\0')
14:         *buffer++ = *arg++;
15:     } else if (*arg == '[') {                            12*
16:       arg++;                                             13
17:       i = *arg++;                                        14
18:       if (*arg++ != '-') {                               15!
19:         *buffer++ = '[';
20:         arg -= 2;
21:         continue;
22:       }
23:       ac = *arg++;
24:       while (i <= ac) *buffer++ = i++;
25:       arg++;       /* Skip ']' */
26:     } else
27:       *buffer++ = *arg++;
28:   }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                         2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                  4
35:   }                                                      5
36:   ...                                                    6
37:   expand(argv[index++], index);                          7
38:   ...
39: }
```

Cuando KLEE se ejecuta en tr, encuentra un error de desbordamiento de búferen la línea 18 en la Figura 1 y luego produce un caso de prueba concreto (tr [“” “”) que lo golpea. Asumiendo las opcionesde la subsección anterior, KLEE ejecuta tr de la siguiente manera:

1. KLEE construye argumentos de cadena de línea de comando simbólicos cuyo contenido no tiene más restricciones queterminación cero. Luego restringe el número

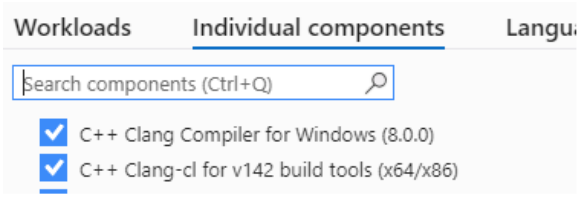
de argumentos a estar entre 0 y 3, y sus tamaños a ser1, 10 y 10 respectivamente. luego llama a main conestas limitaciones de la ruta inicial.

2. Cuando KLEE golpea la rama argc> 1 en la línea 33,utiliza su solucionador de restricciones STP [23] para ver qué direcciones se pueden ejecutar dada la condición de la ruta actual.Para esta rama, ambas direcciones son posibles; KLEEbifurca la ejecución y sigue ambos caminos, agregando la restricción argc> 1 en la ruta falsa y argc ≤ 1en el verdadero camino.
3. Dada más de una ruta activa, KLEE debe elegircuál ejecutar primero. Describimos su algoritmoen la Sección 3.4. Por ahora supongamos que sigue el caminoque llega al bicho. Mientras lo hace, KLEE agrega másrestricciones al contenido de arg, y bifurcaciones para untotal de cinco veces (líneas indicadas con un "*"): dos vecesen la línea 33, y luego en las líneas 3, 4 y 15 en expandir.
4. En cada operación peligrosa (por ejemplo, desreferencia del puntero), KLEE verifica si hay algún valor posible permitido porla condición de la ruta actual causaría un error. Enla ruta anotada, KLEE no detecta errores antes de la línea18. En ese momento, sin embargo, determina que la entradaexiste valores que permiten que la lectura de arg salga delímites: después de tomar la rama verdadera en la línea 15, el código incrementa arg dos veces sin verificar si la cadena ha terminado. Si es así, este incremento omite elterminando '\ 0' y apunta a una memoria no válida.
5. KLEE genera valores concretos para argc y argv(es decir, tr ["" ""] que cuando se vuelve a ejecutar en una versión sin procesar de tr encontrará este error. Luego continúa siguiendo la ruta actual, agregando la restricción de que élno se produce el error (para encontrar otros errores).

Prueba del primer tutorial de Klee – Una pequeña función

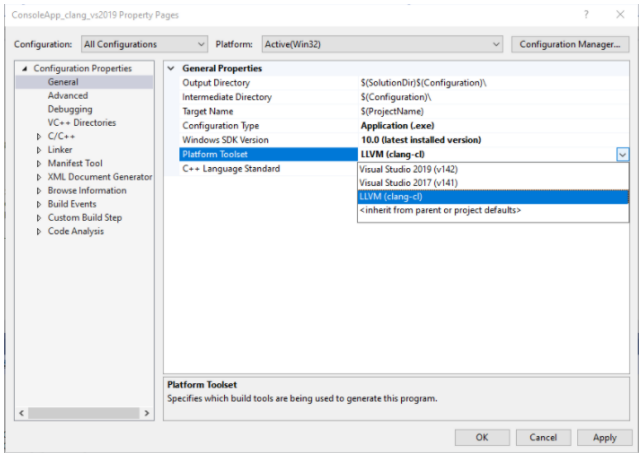
Instalar en pc

Para obtener la mejor compatibilidad con IDE en Visual Studio, recomendamos utilizar las últimas herramientas de compilación de Clang para Windows. Si aún no los tiene, puede instalarlos abriendo el instalador de Visual Studio y eligiendo las herramientas C ++ Clang para Windows en Desarrollo de escritorio con componentes opcionales de C ++ . Si prefiere utilizar una instalación de Clang existente en su máquina, elija C ++ Clang-cl para las herramientas de compilación v142. componente opcional. La biblioteca estándar de Microsoft C ++ actualmente requiere al menos Clang 8.0.0; la versión empaquetada de Clang se actualizará automáticamente para mantenerse al día con las actualizaciones en la implementación de Microsoft de la Biblioteca estándar.



Configurar un proyecto de Windows para usar herramientas Clang

Para configurar un proyecto de Visual Studio para usar Clang, haga clic con el botón derecho en el nodo del proyecto en el Explorador de soluciones y elija Propiedades . Por lo general, primero debe elegir Todas las configuraciones en la parte superior del cuadro de diálogo. Luego, en General > Conjunto de herramientas de plataforma , elija LLVM (clang-cl) y luego Aceptar .



Si está utilizando las herramientas de Clang que se incluyen con Visual Studio, no se requieren pasos adicionales. Para proyectos de Windows, Visual Studio invoca de forma predeterminada a Clang en modo clang-cl y se vincula con la implementación de Microsoft de la biblioteca estándar. De forma predeterminada, clang-cl.exe se encuentra en *% VCINSTALLDIR% \ Tools \ Llvm \ bin * y *% VCINSTALLDIR% \ Tools \ Llvm \ x64 \ bin * .

Si está utilizando una instalación personalizada de Clang, puede modificar Proyecto > Propiedades > DIrectorios VC ++ > Propiedades de configuración > Directorios ejecutables agregando la raíz de instalación personalizada de Clang como el primer directorio allí, o cambiar el valor de la LLVMInstallDirpropiedad. Consulte Establecer una ubicación LLVM personalizada para obtener más información.

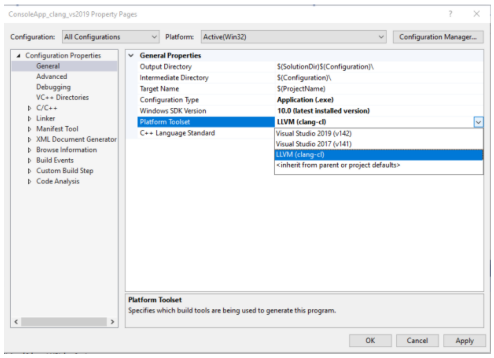
Configurar un proyecto de Linux para usar herramientas Clang

Para proyectos de Linux, Visual Studio usa la interfaz compatible con Clang GCC. Las propiedades del proyecto y casi todos los indicadores del compilador son idénticos

Para configurar un proyecto de Visual Studio Linux para usar Clang:

1. Haga clic con el botón derecho en el nodo del proyecto en el Explorador de soluciones y elija Propiedades.

2. Por lo general, primero debe elegir Todas las configuraciones en la parte superior del cuadro de diálogo.
3. En General > Conjunto de herramientas de plataforma , elija WSL_Clang_1_0 si está usando el Subsistema de Windows para Linux, o Remote_Clang_1_0 si está usando una máquina o VM remota.
4. Presione OK .



En Linux, Visual Studio usa de forma predeterminada la primera ubicación de Clang que encuentra en la propiedad del entorno PATH. Si está utilizando una instalación personalizada de Clang, debe cambiar el valor de la LLVMInstallDirpropiedad o sustituir una ruta en Proyecto > Propiedades > DIrectorios VC ++ > Propiedades de configuración > Directorios ejecutables . Consulte Establecer una ubicación LLVM personalizada para obtener más información.

Establecer una ubicación LLVM personalizada

Puede establecer una ruta personalizada para LLVM para uno o más proyectos creando un archivo *Directory.build.props* y agregando ese archivo a la carpeta raíz de cualquier proyecto. Puede agregarlo a la carpeta raíz de la solución para aplicarlo a todos los proyectos de la solución. El archivo debería verse así (pero sustituya su ruta real):

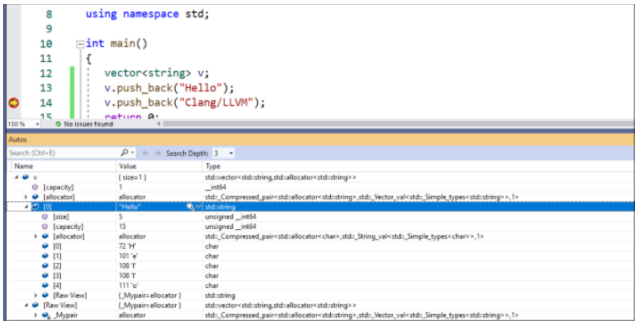
XML Copiar

```
<Project>
  <PropertyGroup>
    <LLVMInstallDir>c:\MyLLVMRootDir</LLVMInstallDir>
  </PropertyGroup>
</Project>
```

Establezca propiedades adicionales, edite, compile y depure

Después de haber establecido una configuración de Clang, haga clic derecho nuevamente en el nodo del proyecto y elija Recargar proyecto . Ahora puede compilar y depurar el proyecto con las herramientas de Clang. Visual Studio detecta que está utilizando el compilador de Clang y proporciona IntelliSense, resaltado, navegación y otras funciones de edición. Los errores y advertencias se muestran en la ventana de resultados . Las páginas de propiedades del proyecto para una configuración de Clang son muy similares a las de MSVC, aunque algunas características que dependen del compilador, como Editar y Continuar, no están disponibles para las configuraciones de Clang. Para configurar un compilador de Clang o una opción de enlazador que no esté disponible en las páginas de propiedades, puede agregarlo manualmente en las páginas de propiedades en Propiedades de configuración > C / C ++ (o enlazador)> Línea de comando > Opciones adicionales .

Al depurar, puede utilizar puntos de interrupción, visualización de datos y memoria, y la mayoría de las demás funciones de depuración.



Prueba del primer tutorial de Klee – Una pequeña función

Este tutorial lo guía a través de los pasos principales necesarios para probar una función simple con KLEE. Aquí está nuestra función simple:

```
intget_sign(intx){
if(x==0)
return0;

if(x<0)
return-1;
else
return1;
```

}

Puede encontrar el código completo para este ejemplo en el árbol de fuentes debajo examples/get_sign.

Marcar la entrada como simbólica

Para probar esta función con KLEE, necesitamos ejecutarla en *una* entrada *simbólica* . Para marcar una variable como simbólica, usamos la klee_make_symbolic()función (definida en klee/klee.h), que toma tres argumentos: la dirección de la variable (ubicación de memoria) que queremos tratar como simbólica, su tamaño y un nombre (que puede ser cualquier cosa) . Aquí hay una main()función simple que marca una variable acomo simbólica y la usa para llamar get_sign():

```
intmain(){
inta;
klee_make_symbolic(&a,sizeof(a),"a");
returnget_sign(a);
}
```

Compilación en código de bits LLVM

KLEE opera en código de bits LLVM. Para ejecutar un programa con KLEE, primero compílelo en el código de bits LLVM usando clang -emit-llvm.

Desde dentro del examples/get_signdirectorio:

```
$ clang -I ../../include -emit-llvm-c-g-O0-Xclang-disable-O0-optnone get_sign.c
```

que debería crear un get_sign.bcarchivo en formato de código de bits LLVM. El -I argumento se usa para que el compilador pueda encontrar klee/klee.h, que contiene definiciones para las funciones intrínsecas que se usan para interactuar con la máquina virtual KLEE, como klee_make_symbolic. Es útil construir con -gpara agregar información de depuración al archivo de código de bits, que usamos para generar información de estadísticas de nivel de línea de origen.

El código de bits pasado a KLEE no debe optimizarse, porque seleccionamos las optimizaciones correctas para KLEE que se pueden habilitar con la --optimize opción de línea de comandos de KLEE . Sin embargo, en una versión posterior de las versiones LLVM (> 5.0), la -O0marca cero NO debe usarse al compilar para KLEE, ya que evita que KLEE realice sus propias optimizaciones. -O0 -Xclang -disable-O0-optnoneen su lugar, consulte este número para obtener más detalles.

Si no desea reproducir los casos de prueba como se describe más adelante y no le importa la información de depuración y la optimización, puede eliminar la klee/klee.hinclusión y luego compilar get_sign.ccon:
$ clang **-emit-llvm-c** get_sign.c

Sin embargo, recomendamos usar el comando más largo anterior.

Ejecutando KLEE

Para ejecutar KLEE en el archivo de código de bits, simplemente ejecute:
$ klee get_sign.bc

Debería ver la siguiente salida (asume LLVM 3.4):
KLEE: output directory ="klee-out-0"

KLEE: **done**: total instructions = 31
KLEE: **done**: completed paths = 3
KLEE: **done**: generated tests = 3

Hay tres caminos a través de nuestra función simple, uno donde a es 0, otro donde es menor que 0y otro donde es mayor que 0. Como era de esperar, KLEE nos informa que exploró tres rutas en el programa y generó un caso de prueba para cada ruta explorada. La salida de una ejecución de KLEE es un directorio (en nuestro caso klee-out-0) que contiene los casos de prueba generados por KLEE. KLEE nombra el directorio de salida klee-out-Ndonde Nestá el número más bajo disponible (por lo tanto, si ejecutamos KLEE nuevamente, creará un directorio llamado klee-out-1), y también genera un enlace simbólico llamado klee-lasta este directorio para mayor comodidad:
$ ls klee-last/
assembly.ll run.istats test000002.ktest
info run.stats test000003.ktest
messages.txt test000001.ktest warnings.txt

Haga clic aquí si desea obtener una descripción general de los archivos generados por KLEE. En este tutorial, solo nos enfocamos en los archivos de prueba reales generados por KLEE.

Casos de prueba generados por KLEE

Los casos de prueba generados por KLEE se escriben en archivos con .ktestextensión. Estos son archivos binarios que se pueden leer con la utilidad *ktest-tool* . *ktest-tool* genera diferentes representaciones para el mismo objeto, por ejemplo, cadenas de bytes de Python (datos), enteros (int) o texto ascii (texto). Así que examinemos cada archivo:
$ ktest-tool klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args : ['get_sign.bc']
num objects: 1

```
object 0: name: 'a'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x00'
object 0: hex : 0x00000000
object 0: int : 0
object 0: uint: 0
object 0: text: ....

$ ktest-tool klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['get_sign.bc']
num objects: 1
object 0: name: 'a'
object 0: size: 4
object 0: data: b'\x01\x01\x01\x01'
object 0: hex : 0x01010101
object 0: int : 16843009
object 0: uint: 16843009
object 0: text: ....

$ ktest-tool klee-last/test000003.ktest
ktest file : 'klee-last/test000003.ktest'
args       : ['get_sign.bc']
num objects: 1
object 0: name: 'a'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x80'
object 0: hex : 0x00000080
object 0: int : -2147483648
object 0: uint: 2147483648
object 0: text: ....
```

En cada archivo de prueba, KLEE informa los argumentos con los que se invocó el programa (en nuestro caso, ningún argumento más que el nombre del programa), el número de objetos simbólicos en esa ruta (solo uno en nuestro caso), el nombre de nuestro object ('a') y su tamaño (4). La prueba real en sí está representada por el valor de nuestra entrada: 0para la primera prueba, 16843009para la segunda y -2147483648para la última. Como se esperaba, KLEE generó un valor 0, un valor positivo (16843009) y un valor negativo (-2147483648). ¡Ahora podemos ejecutar estos valores en una versión nativa de nuestro programa para ejercitar todas las rutas a través del código!

Reproducción de un caso de prueba

Si bien podemos ejecutar los casos de prueba generados por KLEE en nuestro programa a mano (o con la ayuda de una infraestructura de prueba existente), KLEE proporciona una *biblioteca de reproducción* conveniente , que simplemente reemplaza la llamada a klee_make_symboliccon una llamada a una función que se asigna a nuestra entrada el valor almacenado en el .ktestarchivo. Para usarlo, simplemente vincule su programa con la libkleeRuntestbiblioteca y configure la KTEST_FILEvariable de entorno para que apunte al nombre del caso de prueba deseado:

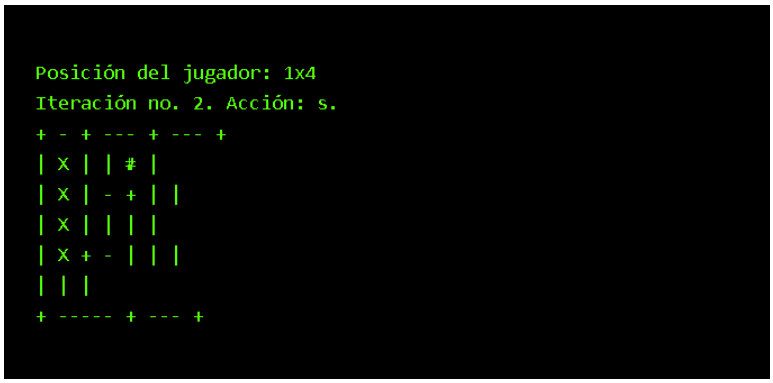
```
$ export LD_LIBRARY_PATH=path-to-klee-build-dir/lib/:$LD_LIBRARY_PATH

$ gcc -I ../../include -L path-to-klee-build-dir/lib/ get_sign.c -lkleeRuntest

$ KTEST_FILE=klee-last/test000001.ktest ./a.out
$ echo$?
0
$ KTEST_FILE=klee-last/test000002.ktest ./a.out
$ echo$?
1
$ KTEST_FILE=klee-last/test000003.ktest ./a.out
$ echo$?
255
```

Como era de esperar, nuestro programa regresa 0cuando se ejecuta el primer caso de prueba, 1 cuando se ejecuta el segundo y 255(-1convertido a un valor de código de salida válido en el 0-255rango) cuando se ejecuta el último.

Prueba del segundo tutorial – Resolviendo un laberinto

El laberinto fue programado inicialmente en c, y se utilizo klee para probarlo y explorar todos los caminos posibles. Código en c: https://pastebin.com/6wG5stht

Este simple juego ASCII te pide que primero lo alimentes con instrucciones. Debe ingresarlos como una lista de acciones por lotes. Como siempre"; a es izquierda, d es derecha, w es arriba y s es abajo. Tiene este aspecto ...



La idea es utilizar klee para resolver este pequeño puzzle.

En la función principal hay variables locales para mantener la posición del "jugador", el contador de iteraciones y una matriz de 28 bytes de las acciones ...

```
48. int
49. main (int argc, char *argv[])
50. {
51.     int x, y;     //Player position
52.     int ox, oy;   //Old player position
53.     int i = 0;    //Iteration number
54.     #define ITERS 28
55.     char program[ITERS];
56.
```

La posición inicial del jugador se establece en (1,1), la primera celda libre en el mapa. Y el jugador 'sprite' es la letra 'X' ...

```
57. //Initial position
58.     x = 1;
59.     y = 1;
60.     maze[y][x]='X';
```

¡En este punto estamos listos para comenzar! Entonces pide direcciones. Lee todas las acciones a la vez como una matriz de caracteres. Ejecutará hasta iteraciones o comandos de ITERS.

```
71. //Read the directions 'program' to execute...
72.     read(0,program,ITERS);
73.
```

Diferentes acciones cambian la posición del jugador en los diferentes ejes y direcciones. Como siempre"; **a** es izquierda, **d** es derecha, **w** es arriba y **s** es abajo.

```
80.         //Move polayer position depending on the actual command
81.         switch (program[i])
82.           {
83.           case 'w':
84.               y--;
85.               break;
86.           case 's':
87.               y++;
88.               break;
89.           case 'a':
90.               x--;
91.               break;
92.           case 'd':
93.               x++;
94.               break;
95.           default:
96.               printf("Wrong command!(only w,s,a,d accepted!)\n");
97.               printf("You loose!\n");
98.               exit(-1);
99.           }
```

¡Comprueba si el premio ha sido acertado! Si es afirmativo… ¡Usted gana!

```
101.          //If hit the price, You Win!!
102.          if (maze[y][x] == '#')
103.            {
104.                printf ("You win!\n");
105.                printf ("Your solution <%42s>\n",program);
106.                exit (1);
107.            }
```

Si algo va mal, no avance, ¡retroceda al estado guardado!

```
108.          //If something is wrong do not advance
109.          if (maze[y][x] != ' '
110.              &&
111.              !((y == 2 && maze[y][x] == '|' && x > 0 && x < W)))
112.            {
113.                x = ox;
114.                y = oy;
115.            }
```

¡Si choca con una pared o si no puede moverse! ¡Sal, pierdes!

```
121.          //If crashed to a wall! Exit, you loose
122.          if (ox==x && oy==y){
123.                printf("You loose\n");
124.                exit(-2);
125.          }
```

Básicamente si podemos movernos ... ¡nos movemos! Coloca al jugador en la posición correcta en el mapa. Y dibuja el nuevo estado.

```
126.          //put the player on the maze...
127.          maze[y][x]='X';
128.          //draw it
129.          draw ();
```

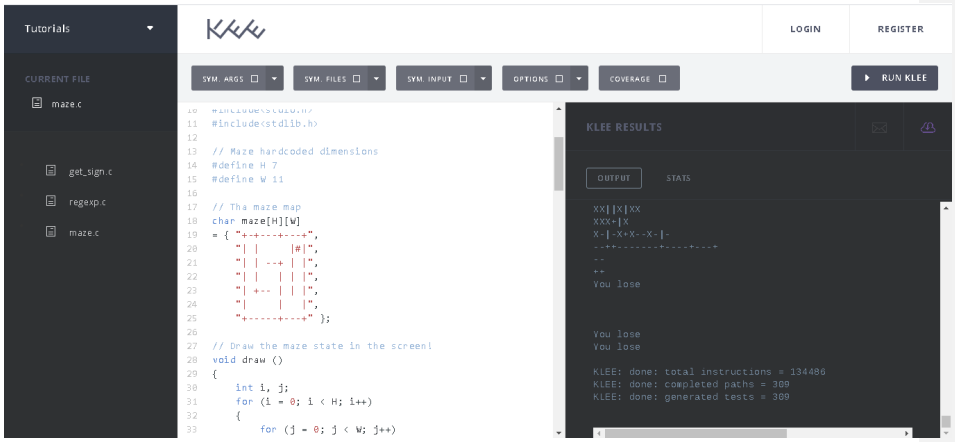
Aquí un ejemplo de como ganar de la forma más obvia posible:

Motor de ejecución simbólica

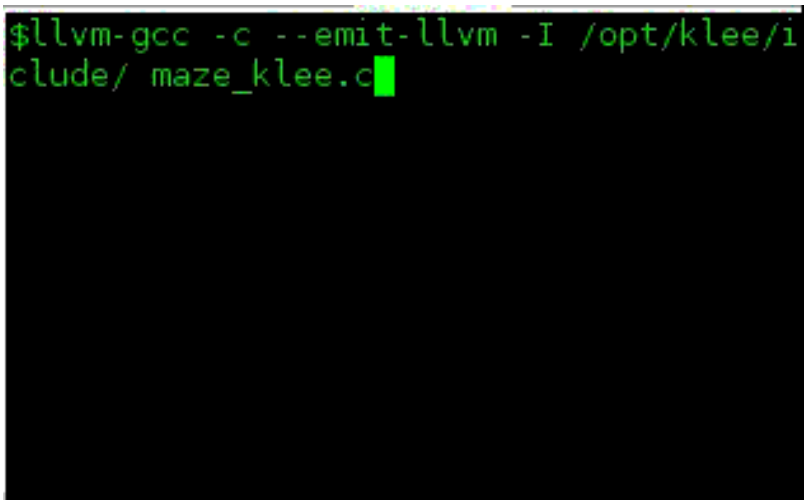
Ahora KLEE encontrará todos los códigos / rutas de laberinto posibles accesibles desde cualquier entrada. Si algunas de esas rutas conducen a una condición de error típica, como una falla de memoria o algo así, ¡KLEE lo indicará!

Motor de ejecución simbólica

Aquí una captura de pantalla de klee ejecutando el laberinto en la página de klee:



Aquí klee nos dice que encontró 309 rutas diferentes, sin embargo, ejecutando el código klee encuentra 321 rutas diferentes



Cada caso de prueba podría recuperarse con la herramienta ktest como esta ...

```
$ ktest-tool klee-last / test000222.ktest
archivo ktest: 'klee-last / test000222.ktest'
args: ['maze_klee.o']
num objetos: 1
objeto 0: nombre: 'programa'
objeto 0: tamaño: 29
objeto 0: datos: 'sssddddwwaawwddddssssddwwwd \ x00'
```

Ok, hasta ahora todo bien, pero no estoy probando todos los casos de prueba posibles y verifico si es una solución de laberinto. Necesitamos una forma para que KLEE nos ayude a diferenciar los casos de prueba normales de los que realmente llegan al "¡Tú ganas!" estado. Tenga en cuenta también que KLEE no ha encontrado ningún error en el código del laberinto. Por diseño, KLEE emitirá una advertencia cuando se detecte cualquier condición de error "bien conocida" (como un acceso a la memoria indexado incorrectamente).

¿Cómo marcar la parte del código que nos interesa?

Hay una función klee_assert () que prácticamente hace lo mismo que una afirmación común de C, obliga a que una condición sea verdadera, de lo contrario aborta la ejecución. Puede consultar la interfaz completa de KLEE C aquí . Pero ya tenemos lo que necesitamos ... una forma de marcar cierta parte del programa (con una afirmación) para que KLEE grite cuando la alcance.

En el código, eso se hace reemplazando esta línea ...

```
printf ("¡Tú ganas! \ n");
```

Por estas dos:

```
printf ("¡Tú ganas! \ n");
klee_assert (0); // Señal ¡¡La solución !!
```

Ahora KLEE afirmará una falla sintética cuando alcance el estado "Usted gana" (eso significa que el 'jugador' presionó el '#). De acuerdo, si lo compila en LLVM y ejecuta KLEE en la nueva versión, marca un caso de prueba como siendo también un error ...

```
$ ls -1 klee-last / | grep -A2 -B2 err
test000096.ktest
test000097.ktest
test000098. asert.err
test000098.ktest
test000098.pc
```

Veamos cuál es la entrada que desencadena esta solución de error / laberinto ...

```
$ ktest-tool klee-last / test000098.ktest
archivo ktest: 'klee-last / test000098.ktest'
args: ['maze_klee.o']
num objetos: 1
objeto 0: nombre: 'programa'
objeto 0: tamaño: 29
objeto 0: datos: 'sddwddddssssddwwww \ x00 \ x00 \ x00 \ x00 \
x00 \ x00 \ x00 \ x00 \ x00 \ x00 \ x00'
```

Así que propone la solución ...

Sddwddddssssddwwww

Si somos observativos podremos ver dos cosas, la solución parece demasiado corta y por la combinación de teclas inmediatamente después de dar el primer movimiento hacia abajo se desvía a la derecha…

Un muro falso… en un videojuego esto sería un “easter egg”, pero en software empresarial esto se podría definir como un error (como anteriormente se mencionó, klee es un motor de ejecución simbólica, estos fueron inicialmente propuestos para descubrir brechas y repararlas en hacking).

Klee al final ofreció 4 posibles soluciones al laberinto (recuerden que teníamos 321 rutas diferentes…)

```
$ ktest-tool klee-last / test000097.ktest
archivo ktest: 'klee-last / test000097.ktest'
args: ['maze_klee.o']
num objetos: 1
objeto 0: nombre: 'programa'
objeto 0: tamaño: 29
objeto 0: datos: 'sddwddddsddw \ x00 \ x00 \ x00 \ x00 \
x00 \ x00 \ x00 \ x00 \ x00 \ x00 \ x00 \ x00 \ x00 \ x00 \
x00 \ x00 \ x00'
$ ktest-tool klee-last / test000136.ktest
archivo ktest: 'klee-last / test000136.ktest'
args: ['maze_klee.o']
num objetos: 1
objeto 0: nombre: 'programa'
objeto 0: tamaño: 29
objeto 0: datos: 'sddwddddssssddwwww \ x00 \ x00 \ x00 \
x00 \ x00 \ x00 \ x00 \ x00 \ x00 \ x00 \ x00 '
$ ktest-tool klee-last /
test000239.ktest archivo ktest:' klee-last /
test000239.ktest '
args: [' maze_klee.o ']
núm de objetos: 1
objeto 0: nombre: 'programa'
objeto 0: tamaño: 29
objeto 0: datos: 'ssssddddwwaawwddddsddw \ x00 \ x00 \ x00
\ x00 \ x00 \ x00 \ x00'
$ ktest-tool klee-last /
test000268.ktest archivo ktest: 'klee-last /
test000268.ktest'
args: ['maze_klee.o']
num objetos: 1
objeto 0: nombre: 'programa'
objeto 0: tamaño: 29
objeto 0: datos: 'ssssddddwwaawwddddssssddwwww \ x00'
```

Hay 4 posibles soluciones:

1. sssddddwwaawwddddssssddwwww
2. sssddddwwaawwddddsddw
3. sddwddddssssddwwww
4. sddwddddsddw

Motor de ejecución simbólica

Prueba de programas Rust con klee

El compilador de Rust se basa en la plataforma LLVM y KLEE se basa en LLVM, por lo que los pasos necesarios para utilizar KLEE para verificar los programas de Rust son:

1. Compile el programa para generar código de bits LLVM.
2. Ejecute KLEE en el archivo de código de bits.
3. Examine la salida de KLEE.

Un pequeño programa de prueba

Como ejemplo continuo, usaremos el mismo ejemplo que usamos para explicar cómo usar la caja de anotaciones de verificación .

Este código está en demos/simple/kleey los comandos de shell en este archivo están en demos/simple/klee/verify.sh.

utilice las anotaciones_de_verificación como verificador;

```
fn  principal () {
    dejar un: U32  = verificador :: AbstractValue :: abstract_value ();
   deje b: u32  = verificador :: AbstractValue :: abstract_value ();
   verificador :: asumir ( 1  <= a && a <=  1000 );
   verificador :: asumir ( 1  <= b && b <=  1000 );
   Si verificador :: is_replay () {
       eprintln! ( "Valores de prueba: a = {}, b = {}" , a, b);
   }
   sea r = a * b;
   verificador :: ¡afirmar! ( 1  <= r && r <=  1000000 );
}
```

El compilador de Rust y KLEE están en el Dockerfile (ver instalación), así que inicie la imagen de Docker ejecutando

demostraciones en cd / simple / klee

Docker / ejecutar

Todos los comandos restantes en este archivo se ejecutarán en esta imagen acoplable.

(Por lo general, es más fácil ejecutar esto en una terminal mientras se usa un editor separado para editar los archivos en otra terminal).

Motor de ejecución simbólica

Compilar Rust para verificación

El compilador de Rust funciona en cuatro etapas: primero, compila el código fuente de Rust en HIR (el IR de alto nivel); HIR se convierte en MIR (el IR de nivel medio); MIR se convierte a LLVM IR (el IR de máquina virtual de bajo nivel); y, finalmente, compila LLVM IR hasta el código de máquina y vincula el resultado.

Cuando utilizamos una herramienta basada en LLVM como KLEE, necesitamos modificar este comportamiento para que LLVM IR esté vinculado y guardado en un archivo. Esto se puede hacer pasando banderas adicionales rustccuando se usa cargo para instruir rustcpara vincular los archivos de código de bits.

export RUSTFLAGS="-Clto -Cembed-bitcode=yes --emit=llvm-bc $RUSTFLAGS"

También tenemos que pasar algunas banderas de configuración cargoy rustcconfigurar la verification-annotationscaja correctamente.

export RUSTFLAGS="--cfg=verify $RUSTFLAGS"

Nuestro objetivo es encontrar errores, por lo que activamos alguna comprobación de errores adicional en el código compilado para detectar desbordamientos aritméticos.

export RUSTFLAGS="-Warithmetic-overflow -Coverflow-checks=yes $RUSTFLAGS"

Y, cuando encontramos un error, queremos informarlo de la manera más eficiente posible para asegurarnos de que el programa abortará si entra en pánico.

export RUSTFLAGS="-Zpanic_abort_tests -Cpanic=abort $RUSTFLAGS"

Con todas esas definiciones, ahora podemos ejecutar

cargo build --features=verifier-klee

Dependiendo de la plataforma en la que esté ejecutando, el archivo de código de bits LLVM resultante se puede colocar en target/debug/deps/try-klee.bc(OSX) o en un archivo con un nombre como target/debug/deps/try_klee-6136b0f50ac42b91.bc(Linux). Al menos para casos simples, podemos referirnos a ambos archivos como target/debug/deps/try_klee*.bcy no preocuparnos por el nombre exacto del archivo.

Lo anterior debería haber funcionado. Pero, si produjo un error de vinculación que involucre, -lkleeRuntestes posible que deba apuntar el vinculador al directorio donde se instaló la biblioteca KLEE. Por ejemplo, en OSX es posible que lo haya instalado $HOME/homebrew/liby necesite usar este comando

export RUSTFLAGS="-L$HOME/homebrew/lib -C link-args=-Wl,-rpath,$HOME/homebrew/lib $RUSTFLAGS"

Compilar programas grandes

Finalmente, en proyectos más grandes y complejos que este ejemplo, hemos visto problemas con el compilador de Rust que genera instrucciones SSE que KLEE no admite. No tenemos una solución completa para esto, pero hemos descubierto que ayuda compilar con un nivel bajo (pero distinto

de cero) de optimización, para deshabilitar optimizaciones más sofisticadas y para deshabilitar las funciones SSE y AVX.

export RUSTFLAGS="-Copt-level=1 $RUSTFLAGS"

export RUSTFLAGS="-Cno-vectorize-loops -Cno-vectorize-slp $RUSTFLAGS"

export RUSTFLAGS="-Ctarget-feature=-mmx,-sse,-sse2,-sse3,-ssse3,-sse4.1,-sse4.2,-3dnow,-3dnowa,-avx,-avx2 $RUSTFLAGS"

(Esto no es necesario para nuestro ejemplo simple).

Ejecutando KLEE

Habiendo creado un archivo de código de bits que contiene el programa y todas las bibliotecas de las que depende, ahora podemos ejecutar KLEE así:

klee --libc=klee --silent-klee-assume --output-dir=kleeout --warnings-only-to-file target/debug/deps/try_klee*.bc

Este comando producirá un resultado como este

KLEE: output directory is "try-klee/kleeout"

KLEE: Using STP solver backend

... (possibly warnings about different target triples - probably benign)

KLEE: done: total instructions = 10153

KLEE: done: completed paths = 3

KLEE: done: generated tests = 1

(En OSX, ¿también puede fallar y producir un volcado de pila después de producir esa salida?)

Esto muestra que KLEE exploró tres rutas a través del código anterior, encontró una ruta que falló y generó un archivo que contiene entradas que pueden activar esa ruta fallida. Para encontrar esos valores de entrada, buscamos en el directorio kleeout archivos con nombres como test000001.ktest. Estos son archivos binarios que se pueden examinar con KLEEktest-tool

$ ktest-tool kleeout/test000001.ktest

ktest file : 'kleeout/test000001.ktest'

args : ['target/debug/deps/try_klee.bc']

num objects: 2

object 0: name: 'unnamed'

object 0: size: 4

Motor de ejecución simbólica

object 0: data: b'\x01\x00\x00\x00'

object 0: hex : 0x01000000

object 0: int : 1

object 0: uint: 1

object 0: text: ....

object 1: name: 'unnamed'

object 1: size: 4

object 1: data: b'\x01\x00\x00\x00'

object 1: hex : 0x01000000

object 1: int : 1

object 1: uint: 1

object 1: text: ....

Esto demuestra que el programa crea dos objetos no deterministas cada uno llamado 'unnamed` y cada uno de tamaño 4. KLEE no sabe la manera de interpretar lo que los displays de ellos varias interpretaciones comunes.

Reproducir los valores de entrada

Cuando compilamos el programa para producir código de bits LLVM, también generamos un binario convencional. Podemos ejecutar esto de la manera normal:

cargo run --features=verifier-klee

y KLEE nos pedirá el nombre del archivo de entrada

KLEE-RUNTIME: KTEST_FILE not set, please enter .ktest path:

O podemos especificar el archivo de entrada cuando invocamos a KLEE:

$ KTEST_FILE=kleeout/test000001.ktest cargo run --features=verifier-klee

Finished dev [unoptimized + debuginfo] target(s) in 0.00s

Running `target/debug/try-klee`

Test values: a = 1, b = 1

Esto proporciona una manera más fácil de ver los valores de prueba elegidos: utilizando la función de impresión incorporada de Rust.

Manejo de programas más grandes

Motor de ejecución simbólica

Las instrucciones anteriores funcionan bien para programas pequeños. Pero, a medida que abordamos programas con dependencias, necesitamos agregar algunos indicadores más para que todo funcione.

Compilar programas más grandes

Si agrega la siguiente dependencia en serde

serde = { version = "1.0", features = ["derive"] }

es probable que obtenga un error de enlace

error: cannot prefer dynamic linking when performing LTO

La solución (increíblemente oscura) para esto es especificar un objetivo explícitamente. (¡No tengo idea de por qué esto ayuda!)

El primer paso es averiguar qué objetivo está utilizando actualmente. Quieres que el "host predeterminado" informe por "rustup show"

$ rustup show | grep Default

Default host: x86_64-unknown-linux-gnu

Ahora que conocemos el objetivo, agregamos --targetal cargo buildcomando

cargo build --features=verifier-klee --target=x86_64-unknown-linux-gnu

Desafortunadamente, esto cambia donde se coloca el archivo de código de bits final. En lugar de target/debug/deps/try_klee*.bc eso, se coloca target/x86_64-unknown-linux-gnu/debug/deps/try_klee*.bc.

Usar KLEE con programas más grandes

En lo anterior, usamos este comando para invocar KLEE

klee --libc=klee --output-dir=kleeout --warnings-only-to-file target/debug/deps/try_klee*.bc

Algunas banderas adicionales que vale la pena usar para ejemplos más grandes son

- --exit-on-errorhace que KLEE se cierre tan pronto como encuentre un problema. (El modo predeterminado de KLEE es seguir buscando más problemas).
- --disable-verifysoluciona un problema en KLEE causado por una interacción entre la información de depuración y la inserción .

Todos estos indicadores se agregan antes del archivo de código de bits como este

```
klee --output-dir=kleeout --warnings-only-to-file --exit-on-error \
    --libc=klee --silent-klee-assume --disable-verify \
    target/x86_64-unknown-linux-gnu/debug/deps/try_klee*.bc
```

Bibliografía

1. Página principal de klee- http://klee.doc.ic.ac.uk/#
2. Página con el código del ejemplo del laberinto - https://pastebin.com/6wG5stht
3. Página del tutorial ejecutado del laberinto - https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/
4. Página con todos los tutoriales de klee -http://klee.github.io/tutorials/
5. Documentación de klee - https://www.usenix.org/legacy/event/osdi08/tech/full_papers/cadar/cadar.pdf
6. Pdf sobre la explicación de las técnicas de análisis en general https://repositorio.cbachilleres.edu.mx/wp-content/material/compendios/cuarto/tec_analisis.pdf
7. ¿Qué es una técnica de análisis?- http://fccea.unicauca.edu.co/old/tgarf/tgarfse103.html#:~:text=En%20la%20t%C3%A9cnica%20del%20An%C3%A1lisis,no%20pueden%20omitirse%20sus%20relaciones.
8. Opinión sobre la ejecución simbólica encontrada en un blog - https://s3lab.deusto.es/idea-ejecucion-simbolica/
9. Video tutoriales de los primeros 4 ejemplos encontrados en la página (desde como instalarlo hasta como ejecutarlo) - https://adalogics.com/blog/symbolic-execution-with-klee
10. Tic como entorno simbólico - http://e-ducacion.info/educacion-social-y-tics/caracteristicas-de-las-tic-como-entorno-simbolico-y-sus-potencialidades-para-el-aprendizaje/#:~:text=Caracter%C3%ADsticas%20de%20las%20TIC%20como%20entorno%20simb%C3%B3lico%20y%20sus%20potencialidades%20para%20el%20aprendizaje,-Posted%20by%20Noemi&text=Formalismo%3A%20Se%20refiere%20a%20la,y%20planificaci%C3%B3n%20de%20las%20acciones.
11. Definición de entorno - https://dle.rae.es/entorno
12. Definición de simbólico - https://definicion.mx/simbolico/
13. Pruebas de caja blanca, ¿Qué son y para que se usan en informática? - https://es.wikipedia.org/wiki/Pruebas_de_caja_blanca
14. Pruebas de caja negra, ¿Qué son y para que se usan en informática? - https://es.wikipedia.org/wiki/Caja_negra_(sistemas)