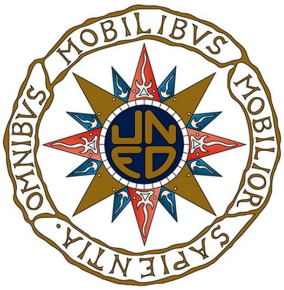
Universidad Nacional de Educación a Distancia
E.T.S de Ingeniería Informática

Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

Código 31105151 - Trabajo Fin de Máster



La Ejecución Simbólica en el Análisis de Vulnerabilidades de Programas

Estudiante: Juan Antonio Osés Aragón
Director: José Antonio Cerrada Somolinos
Curso 2017/2018 - Convocatoria de septiembre

Departamento de Ingeniería de Software y Sistemas Informáticos

UNED - E.T.S de Ingeniería Informática

Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

Código 31105151 - Trabajo Fin de Máster

La Ejecución Simbólica en el Análisis de Vulnerabilidades de Programas

Trabajo Fin de Máster Tipo B

Estudinate: **Juan Antonio Osés Aragón**
Director: **José Antonio Cerrada Somolinos**

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MASTER

Fecha: 08/09/2018

Quién suscribe:

Autor(a): JUAN ANTONIO OSÉS ARAGÓN
D.N.I/N.I.E/Pasaporte.: 33441274X

Hace constar que es la autor(a) del trabajo:

La Ejecución Simbólica en el Análisis de Vulnerabilidades de Programas

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

- ✓ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.
- ✓ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.
- ✓ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Fdo. 

IMPRESO TFDM05_AUTORPBL
AUTORIZACIÓN DE PUBLICACIÓN
CON FINES ACADÉMICOS

UNED ETS de Ingeniería Informática

Impreso TFdM05_AutorPbl. Autorización de publicación y difusión del TFM para fines académicos

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del/los Autor/es

Juan del Rosal, 16
28040, Madrid

Tel: 91 398 89 10
Fax: 91 398 89 09

www.issi.uned.es

Resumen

Las metodologías de detección de vulnerabilidades de software es uno de los campos más activos de investigación para la mejora de la seguridad. En el presente trabajo se analiza el papel que juega la ejecución simbólica dentro de este ámbito. El trabajo comienza con una puesta en contexto de las herramientas de búsqueda de vulnerabilidades y generación de *exploits*, y el uso que estas herramientas hacen de la ejecución simbólica. Seguidamente, y antes de describir formalmente la ejecución simbólica, se hace una presentación de los solucionadores SMT, claves en la aplicación de esta técnica.

En una segunda parte, se presenta la herramienta de ejecución simbólica KLEE, muy utilizado para pruebas de concepto en investigación. A continuación, se estudia las modificaciones de KLEE que faciliten la implementación de un marco de trabajo de test, con el objetivo de su aplicación en el testeo de software.

Finalmente, se presentan las conclusiones obtenidas del estudio y las posibles líneas futuras de investigación.

Palabras clave

Ejecución simbólica, vulnerabilidades de software, *exploits*, solucionadores SMT, KLEE, testeo de programas, análisis binario, seguridad de software.

Índice general

Índice de figuras

Capítulo 1

Introducción

La seguridad del software es uno de los temas fundamentales dentro del campo de la seguridad informática. Los errores de software han sido en numerosas ocasiones convenientemente explotados como fuente de importantes amenazas de seguridad. Así, la creación de *exploits*, una entrada al programa manipulada para provocar una ejecución arbitraria o maliciosa del programa, es uno de los mecanismos más usados para introducirse en los sistemas informáticos y causar daño.

Un reflejo de la importancia de la amenaza que representan estos errores ha quedado patente en la organización de competiciones para la detección de vulnerabilidades, como es el caso del "**Cyber Grand Challenge**" (CGC) de 2016 organizado por DARPA ("*Defense Advanced Research Projects Agency*"). El objetivo de eventos como este es mostrar los últimos avances en sistemas para la detección automática de vulnerabilidades, la generación de *exploits*, y creación de parches de software. En dicha edición de la competición resultaron vencedores los sistemas Mayhem y Mechanical Phish [1].

Para la creación de las herramientas automáticas de verificación como las anteriormente citadas se han empleado tanto técnicas de análisis estático como la comprobación del modelo ("*Model Checking*") y la ejecución simbólica ("*Symbolic Execution*").

La ejecución simbólica tiene como ventaja, frente a la comprobación del modelo, su mayor efectividad en la búsqueda de vulnerabilidades, puesto que las fórmulas que genera crecen más despacio que en el caso de la comprobación del modelo, a medida

que se explora un programa. Además, puede usarse para verificar parcialmente un programa, dada su capacidad para generar casos de test a partir de las rutas que han podido completarse.

El objetivo del presente trabajo es realizar un análisis de las posibilidades de la ejecución simbólica para explorar las vulnerabilidades del software y el soporte que esta proporciona en la generación de casos de test, que sirven como base para la generación de *exploits*.

Por otro lado, la aplicación práctica de esta técnica para la evaluación de vulnerabilidades de programas y la mejora de la calidad es aún limitada, estando la mayoría de las herramientas existentes restringidas a un uso académico o de investigación.

En este trabajo se ha tomado el motor simbólico de código abierto KLEE como base para un realizar el estudio. Se trata de una herramienta orientada a programas en lenguaje C, muy utilizada para pruebas de concepto. Se proponen y estudian varias modificaciones sobre la herramienta KLEE que faciliten su aplicación en las etapas de test de seguridad.

Finalmente, se presentan las conclusiones del estudio y los resultados obtenidos de las modificaciones hechas. Como último punto, se proponen las líneas futuras posibles de investigación en este campo.

Capítulo 2

Vulnerabilidades y *exploits*

Gran parte de los programas actuales han sido escritos en lenguajes como C y C++. Lenguajes que permiten gran libertad al desarrollador proporcionando un control a bajo nivel de las asignaciones de memoria, pero que constituye una fuente de vulnerabilidades en el software construido, que pueden ser explotadas por un atacante.

2.1. Vulnerabilidades de software

De manera formal, una **vulnerabilidad** de un programa binario (P) es un error en P que puede ser usado por un atacante para alterar la ejecución del programa, esto es, violar una determinada política de seguridad (ϕ), explícita o implícita, con el fin de conseguir objetivos dañinos.

Una política de seguridad (ϕ) se modela como un predicado de lógica booleana de primer orden dentro del espacio de estados del programa, y puede tomar el valor "*seguro*" o "*inseguro*". Si una condición de vulnerabilidad (c) viene dada por el valor "*inseguro*", y un punto vulnerable (i_p) es aquel con la primera instrucciíon que satisface la condición c, una vulnerabilidad queda completamente descrita por la tupla $\langle P, i_p, c \rangle$.

Entre las vulnerabilidades más frecuentes y, por tanto, principales objetivos de un sistema de detección, están:

- "*Buffer overflow*": aparece cuando se desborda la escritura en "*buffers*" de tamaño fijo, produciendo la corrupción de los datos, la caída de la aplicación

o la posibilidad de reemplazo de código en el *stack* por parte del atacante.

- Formato de cadenas ("*Format String*"): Se produce cuando una cadena formateada se proporciona en la entrada del programa y termina por tratarse como un comando.

- Corrupción general de la memoria: Ocurre cuando los datos dentro de un "*buffer*" son modificados y esta modificación puede conducir a un fallo de la aplicación. El "*buffer overflow*", el uso de punteros sin inicializar o cuando ya se han liberado, son ejemplos de esta vulnerabilidad.

2.2. Técnicas de análisis

Con el fin de crear software más seguro se han desarrollado distintas técnicas de análisis de programas para vérificar la existencia de vulnerabilidades, con el objetivo de poder ser automatizadas. Técnicas que pueden dividirse en dos categorías: análisis de código fuente y análisis binario.

2.2.1. Análisis del código fuente

El **análisis de código fuente**, (SCA- *"Source Code Analysis"*), se basa en el análisis del código fuente original sin compilar. Consiste principalmente en implementar reglas que deben verificarse a lo largo de todo el código. Aún cuando los modernos compiladores incluyen algunas capacidades de análisis estático, existen herramientas especializadas, como Coverity o Fortify.

Dependiendo del modelado, esta técnica puede detectar desde fallos sencillos a otros mucho más complejos que necesitan del estudio de relaciones más complejas en el código, como puede ser la relación entre distintos archivos fuente. Un ejemplo de implementación de un analizador estático puede consultarse en [2].

Por otro lado, los analizadores de código estático pueden dividirse en dos clases:

- Interprocedural: Utiliza las llamadas entre funciones para crear un modelo estático de llamadas y poder simular la ejecución.

- Intraprocedural: Se centra en el análisis individualizado de cada función.

En general, esta técnica puede calificarse como rápida, de bajo coste y generalmente efectiva. Sin embargo presenta limitaciones como la necesidad de analizadores distintos según el lenguaje, o el alto número de falsos positivos y negativos.

2.2.2. Análisis binario

El **análisis binario o *byte-code*** (BCA - "*Byte-Code Analysis*") analiza el código binario (o "*byte-code*") generado por el compilador, ya sea el código objeto o ejecutable. En esta categoría se incluye también el análisis a nivel de las representaciones intermedias del ejecutable que hacen uso de una máquina virtual, caso del "*byte-code*". En este caso se parte de las instrucciones efectivas, realizando un análisis de bajo nivel en términos de instrucciones, registros, procedimientos o localizaciones de memoria.

El análisis binario presenta algunas ventajas frente al análisis de código fuente, entre las que cabe destacar:

1. Todo el programa puede ser analizado, incluyendo las librerías enlazadas, sin necesidad del empleo de modelos.

2. Los binarios contienen detalles específicos muy importantes que solo están disponibles en tiempo de ejecución: asignación de memoria, uso de los registros y orden de ejecución, entre otros.

3. Si el ejecutable ha sido modificado tras la compilación, dichos cambios son detectables por las herramientas de análisis binario.

4. No es necesario disponer del código fuente del programa.

5. Al contrario que las herramientas de análisis de código que deben soportar distintos lenguajes de programación, una herramienta de análisis binario solo necesita soportar uno o un conjunto muy reducido (normalmente lenguajes intermedios).

6. Si en el código fuente del programa se han insertado instrucciones en ensamblador, estas no necesitan ser tratadas de una forma distinta por la herramienta de análisis binario.

Por el contrario, el análisis binario puede ser difícil, y tal como se apunta en [3] presenta algunas complicaciones importantes:

1. Complejidad del modelado de instrucciones, con una semántica difícil, un comportamiento dependiente de sus parámetros y efectos colaterales debidos a los "*flags*" del procesador. Además, existen distintos conjuntos de instrucciones dependiendo de las arquitecturas o versiones de los procesadores.

2. Ausencia de abstracciones de alto nivel que proporcionan los lenguajes de programación, como las funciones, los tipos y las estructuras de datos. Por ejemplo, tampoco cabría hablar propiamente de "*buffers*" sino de posiciones de memoria.

3. Necesidad de un análisis conjunto de la operación, incluso a nivel del núcleo del sistema operativo y de las interacciones entre procesos.

4. Es necesario tener en cuenta también el código malicioso, que muchas veces emplea métodos de ofuscación para burlar el análisis.

Es necesario remarcar que algunas de las técnicas de análisis binario sí parten del código fuente, ya que se compila en un lenguaje intermedio que ser posteriormente analizado por la herramienta.

Finalmente, cabe señalar que dentro del análisis binario existen dos aproximaciones: **análisis binario estático** y **análisis binario dinámico**.

Análisis binario estático

Se trata de un análisis de los binarios de un programa, sin ejecutar dicho programa. Para este fin se procesa el código binario en ensamblador, generando una representación en lenguaje intermedio y construyendo el gráfico de control de flujo (CFG "*Control Flux Graph*").

El CFG es un grafo donde los nodos representan los bloques básicos de instrucciones máquina ($instr_i$) y los arcos representan los posibles puntos de cambio en el control de flujo ($instr_i, intr_j$), es decir, la transferencia del control de la instrucción $instr_i$, a la $instr_j$. Y de esta forma, una **ruta de ejecución** ("*execution path*") es

una secuencia de nodos del CFG, por tanto, el CFG representa todas las rutas de ejecución posibles en el programa.

Un ejemplo de este análisis lo encontramos en Brumley et al. (2007) [4] con una aproximación estática en el cálculo de **firmas de vulnerabilidad**, técnica esta que tiene también una vertiente dinámica. La generación de una firma para una vulnerabilidad permite no solo caracterizarla sino reconocer los *exploits*, consiguiendo no tener falsos positivos. En [4], a partir del binario (P) y la condición de vulnerabilidad (c) genera el CFG y consigue calcular la precondición más débil en las entradas para activar la vulnerabilidad, esto es, la firma de la vulnerabilidad ($S_{<P,c>} = WP(P, c)$). A partir de ahí, se eliminan las variables de entrada y se convierte el predicado resultante en una expresión regular. Esta expresión regular puede aplicarse sobre una entrada al programa para comprobar si se trata de un *exploit*.

Sin embargo, una de las limitaciones de esta metodología está en el alto coste de escalabilidad y el bajo rendimiento. Además, para el tratamiento de los saltos indirectos en el CFG, es decir, de aquellos que dependen de un valor arbitrario, o del contexto de la aplicación, se realizan aproximaciones que pueden llevar a falsos positivos o a la no detección de la vulnerabilidad.

Análisis binario dinámico

En el análisis binario dinámico (DBA - "*Dynamic Binary Analysis*") se examina el comportamiento del programa durante su ejecución en un entorno. Permite una exploración precisa de las rutas que son ejecutadas, pero alcanza en conjunto una menor cobertura de código que el análisis binario estático. Existen diferentes variantes dentro de este tipo de análisis.

Por un lado están las herramientas que requieren una instrumentación del binario antes de su ejecución, como puede ser el caso de herramientas como Valgrind, Purify o Address Satanizer. Esta instrumentación puede ser de dos tipos:

- *Instrumentación binaria estática*, cuando se realiza antes de la ejecución del programa, ya sea en la generación del código objeto o ejecutable. Esta instrumentación es difícil cuando los datos y el código están mezclados o se usan varios módulos, y es imposible en caso del uso de código enlazado.

- *Instrumentación binaria dinámica*, cuando ocurre durante la ejecución del programa, inyectando código de análisis. Si se usa enlazado dinámico, esta inyección debe hacerse después del enlazado. Tiene la ventaja de que el binario no necesita prepararse antes del análisis. Sin embargo, se penaliza el coste de ejecución y su implementación es más compleja.

Por otro lado, el análisis binario dinámico puede realizarse en conjunción con un motor de ejecución simbólica, utilizando la exploración simbólica junto con la resolución de restricciones, o guiando la exploración a través de entradas concretas, lo que se conoce como ejecución concólica.

2.3. *Exploits* de programas

Un ***exploit*** de un programa que tiene una vulnerabilidad, es el proceso de aprovechar dicha vulnerabilidad, normalmente a través de una entrada al programa que lance con éxito la vulnerabilidad. De manera formal, el *exploit* satisface la condición de vulnerabildad (c) en el punto de vulnerabilidad (i_p).

Pero además, el *exploit* debe redirigir el control de flujo hacia la lógica del atacante, de un modo u otro, dependiendo del tipo de ataque. Esto es lo que se conoce como *payload*, es decir, lo que se desea realizar con el *exploit*.

Uno de los *exploits* más típicos es el conocido como "*return-to-stack*", donde por medio del desbordamiento del *stack* en la llamada a una función se consigue sobrescribir la dirección de retorno, de forma que apunte a una dirección del *stack* que permita la ejecución de un *shellcode* inyectado.

Con el fin de paliar la efectividad de los *exploits*, los sistemas operativos actuales incluyen mejoras como [5]:

1. **ASLR ("*Address Space Layout Randomisation*")**: Esta metodología de seguridad combate la posible corrupción de memoria, haciendo aleatoria la dirección de memoria de los objetos y dificultando que un atacante haga referencia directa a la dirección de memoria de dicho objeto. Así, las implementaciones de ASLR introducen aleatoriedad en un subconjunto del *stack*, *heap*, librerías

compartidas y la imagen del programa (en Linux, secciones .text). Sin embargo, muchas veces no introduce el suficiente grado de entropía para impedir que algunos *exploits* logren burlar esta técnica.

2. **DEP/$W \oplus X$ (“*Data Execution Prevention*”)**: Metodología también para la protección de memoria según la cual cada página de memoria debe poder ser escrita o ser ejecutable (“*Write xor Execute*” , $W \oplus X$). De esta forma, si el *exploit* escribe un “*shellcode*” en memoria, esta posición de memoria no puede ser ejecutable. Necesita de un soporte hardware (bit NX - “*no execute*”) y además no siempre es activado por el sistema operativo. Por otro lado, tampoco impide ciertos tipos de ataques que en vez de escribir la memoria reutilizan el código existente.

2.4. Generación de *exploits*

El testeo de software es, sin duda, la técnica más utilizada para la verificación de programas dada su simplicidad y escalabilidad. Sin embargo, automatizar los test presenta limitaciones a la hora de cubrir todos los casos posibles. Es por eso que se considera la posibilidad de generar y usar *exploits* para el testeo de los programas. Los *exploits* constituyen buenos ejemplos de casos de test, y muestran cuándo una vulnerabilidad es explotable por un atacante. Además, pueden servir, en caso de necesidad, para definir la prioridad a la hora de corregir la vulnerabilidad.

Sin embargo, la **generación de *exploits*** manual es difícil y costosa en tiempo, requiere un conocimiento alto de arquitectura de computadores (del sistema operativo o lenguaje ensamblador) y del propio software susceptible de presentar la vulnerabilidad. Es en este contexto donde aparece la **generación automática de exploits** (*AEG - “Automatic Exploit Generation”*) que permite plantear el uso de *exploits* como una técnica de testeo factible.

En Avgerinos et al. (2011) [6] se puede encontrar una definición formal de **generación de *exploits*** de acuerdo a un problema de verificación de programas, donde se busca verificar que el programa es explotable. Este es un planteamiento adecuado con vistas a la automatización. En este contexto, un *exploit* queda definido como una entrada (ε) que satisface la siguiente expresión lógica:

$$\Pi_{bug}(\varepsilon) \wedge \Pi_{exploit}(\varepsilon) = true \tag{2.1}$$

De este modo, el objetivo de la **generación de *exploits*** es verificar si en algún momento se satisface la ecuación anterior. En la cual existen dos predicados, cuyo significado es el siguiente:

- El **predicado de la ruta vulnerable** (Π_{bug}): representa la ruta de una ejecución que ha violado una determinada propiedad de seguridad. Permite dividir el espacio de todas las entradas posibles entre: entradas seguras e inseguras. Este predicado es suficiente para describir las vulnerabilidades del programa.

- El **predicado del *exploit*** ($\Pi_{exploit}$): representa la lógica que el atacante desea introducir y depende de esta. Puede ser provocar una caída del programa ("*crash*") o inyectar un "*shellcode*". Este predicado permite acotar los *exploits* válidos, si existen, dentro de las entradas inseguras.

Como puede observarse, para que la vulnerabilidad sea explotable necesita cumplir el predicado final $\Pi_{bug} \wedge \Pi_{exploit}$. No basta con que exista la vulnerabilidad (Π_{bug} se cumple). Así, la **generación de *exploits*** va un paso más allá de la búsqueda de vulnerabilidades y se centrá en aquellas que representan errores explotables, y por tanto, una amenaza mayor.

La **generación de *exploits*** queda [6], por tanto, definida por un proceso dividido en varias etapas:

1. Realizar un análisis de vulnerabilidades del programa P, ya sea por medio de técnicas estáticas o dinámicas, para conseguir la condición de la vulnerabilidad (Π_{bug}).

2. Construir la condición del exploit ($\Pi_{exploit}$). Es necesario definir el exploit que se quiere utilizar (ej.: tomar un caso concreto de "*shellcode*"), y extraer información de bajo nivel del programa: direcciíon en memoria del *buffer* vulnerable en el *stack*, dirección de retorno de la funciíon vulnerable y el contenido del *stack* justo antes de que el *exploit* sea lanzado. Se trata de conseguir que el *exploit* sea operativo.

3. Conseguir la expresión $\Pi_{bug} \wedge \Pi_{exploit}$, que asegure que se alcance el punto de localizaciíon de la vulnerabilidad (i_p), se cumpla la condición de vulnerabilidad (c) y que el punto de localización contenga un *exploit* bien formado.

4. Resolver la expresión $\Pi_{bug} \wedge \Pi_{exploit}$ para encontrar un *exploit* concreto y poder verificarlo en el programa P. Empíricamente se observa que en muchos casos no es un solo *exploit* el que verifica esta fórmula, sino que existen variantes que la cumplen.

Dentro de las variantes de la AEG puede encontrarse la generación automática de *exploits* a partir de la distribución de los parches de software (APEG - "*Automatic Patch-based Exploit Generation*"). En el caso de la APEG, Brumley et al. (2008) [7], se analiza el riesgo que supone la política de distribuciíon de parches ya que no todos los programas se actualizan simultáneamente. Un parche de software proporciona información sobre las vulnerabilidades corregidas, y así, da pistas al atacante para encontrar un *exploit* del programa no parcheado. Como no todos los programas se actualizan simultáneamente, desde el momento de la distribución del parche, la amenaza sobre los programas no parcheados aumenta. Si bien el periodo de tiempo de actualización de todos los programas es corto, la automatizaciíon de la la generación de *exploits* hace más evidente el riesgo.

Como puede verse, tanto [7] como [6] definen la búsqueda de *exploits* como un problema de verifición de programas. Ambos utilizan el análisis binario dinámico, siendo la ejecución simbólica un elemento clave. También hacen uso del análisis estático. Así en [7] se muestra que la generaciíon de ciertos *exploits* es posible usando tan solo análisis estático y que también puede combinarse con análisis dinámico para otros. Por otra parte, en [6] se usa un sencillo análisis estático para extraer información adicional con la que mejorar la etapa de ejecución simbólica.

2.5. Analizadores binarios

Con el fin de descubrir de forma automática las vulnerabilidades de software, se han desarrollado sistemas que automatizan su búsqueda en programas binarios, y en muchos casos, también, generan los exploits correspondientes. A continuación se enumeran las herramientas de este tipo más destacables.

MAYHEM

Desarrollado por la compañía *ForAllSecure*, un *spin-off* de la CMU ("*Carnegie Mellon University*". Escrito en C++ y OCaml, se trata de un sistema para la búsqueda de vulnerabilidades explotables que trabaja con los binarios. Asegura la naturaleza explotable generando un *exploit* para cada vulnerabilidad encontrada.

MAYHEM implementa la llamada ejecución simbólica híbrida al combinar la ejecución concólica y la simbólica [8]. Gestiona, así, el lanzamiento de varios procesos simultáneos de búsqueda simbólica. Si la memoria está próxima a llegar al límite el ejecutor híbrido interrumpe durante el tiempo necesario la ejecución de los procesos de búsqueda necesarios, guardando su estado y condición de ruta para poder reactivarlos posteriormente.

La arquitectura de MAYHEM se divide en torno a dos componentes[1, 8] un ejecutor concreto (CEC - "*Concrete Executor Client*") y un motor simbólico (SES - "*Symbolic Executor Server*"). El ejecutor CEC es el que realiza la exploración de la rutas al cargar y ejecutar el binario de forma nativa. En este proceso se añade instrumentación al código y se recopila información sobre el estado del programa, como la memoria y los valores de registro. Esta información se pasa después al motor simbólico SES. El motor SES toma dicha información y transforma el binario en lenguaje intermedio BAP IL, pasando después a interpretarlo simbólicamente. A partir de ahí se obtienen dos tipos de fórmulas, las correspondientes a la ruta y las que determinan si un atacante puede ejecutar un *payload* o conseguir el control sobre un puntero. La consistencia de dichas fórmulas se prueba por medio de un solucionador SMT.

Angr

Se trata de un marco de trabajo ("*framework*"), escrito en Python, de código abierto para el análisis binario de plataforma agnóstica, que implementa una serie de técnicas de análsis binario. Entre sus funcionalidades puede encontrarse: desensamblado de binarios, instrumentación de programas, ejecución simbólica o análisis del control de flujo, entre otros.

Fue implementado con el propósito de proporcionar a los investigadores de una plataforma unificada que permita evaluar y comparar la efectividad de las diferentes

técnicas. Entre sus módulos puede destacarse los siguientes:

1. CLE ("*CLE Loads Everything*"): El cargador binario de angr, permite la carga de binarios compatibles POSIX (Linux, FreeBSD) y Windows, entre otros.

2. Lenguage intermedio: Para conseguir la naturaleza agnóstica de la arquitectura es necesario convertir el binario original en un lenguaje intermedio, en este caso, VEX IR, tomado de Valgrind.

3. *SimVex*: Un módulo para representar y modificar el estado del programa.

4. *Claripy*: Una capa de abstracción sobre un solucionador SMT, normalmente Z3.

5. Módulo de análisis: proporciona una forma de empaquetar el análisis en un formato común que facilite su aplicación sobre distintos proyectos.

Mechanical Phish

Desarrollado para el "DARPA Cyber Grand Challenge" por Shellphish, un equipo compuesto por miembros de la Universidad de Santa Barbara y otras instituciones. Aprovecha varias herramientas de código abierto para el descubrimiento de vulnerabilidades y la generación de *exploits*. Combina de manera eficaz la ejecución concólica, simbólica y el empleo de aleatorizadores ("*fuzzers*"). Sus componentes principales son la funcionalidad de búsqueda de vulnerabilidades, basada en angr, y Driller.

Driller es el módulo encargado de dirigir la exploración se compone un motor de ejecución concólica y de un "*fuzzer*", en este caso AFL ("*American Fuzzy Loop*"). Su operación es la siguiente:

1. Llamada inicial al aleatorizador, con casos de test de entrada o sin ellos: Se exploran así un número dado de rutas, hasta el momento en el que el aleatorizador no consigue descubrir más rutas.

2. Una vez exhausto el aleatorizador, Driller llama al motor de ejecución concólica con las rutas encontradas. Esto consigue explorar nuevas rutas y generar las entradas al programa que las origina.

3. Las nuevas entradas conseguidas en la etapa de exploración simbólica se pasan al aleatorizador y mediante la aplicación de mutaciones trata de conseguir nuevas rutas en el programa.

4. El proceso se repite hasta que se encuentra un fallo del programa o no es posible encontrar entradas que representen nuevas rutas.

Avatar

Es un marco de trabajo de análisis binario dinámico para sistemas empotrados, escrito en Python. Compuesto por varios módulos, constituye un marco de arbitraje dirigido por eventos que orquesta el análisis y la comunicación entre un emulador y el sistema físico a analizar. Se basa en la emulación parcial del "*firmware*" mediante S2E, un motor de ejecución simbólica basado en KLEE y QEMU. Las peticiones de entrada y salida que no pueden ser emuladas son redirigidas al sistema empotrado real, ya sea a través de puertos de depuración dedicados o por medio de módulos de software inyectados en el dispositivo.

Avatar ha sido utilizado con éxito en el análisis de vulnerabilidades y la generación de *exploits* en sistemas empotrados, [9]. Sin embargo, en [9] también se señala que Avatar no es escalable para sistemas empotrados complejos, ya que la comunicación entre el motor simbólico y el dispositivo objeto del análisis es muy lenta. Además, se concluye que Avatar carece de un mecanismo para determinar el tipo de vulnerabilidad que se ha producido.

Avatar 2

Se trata del sucesor de Avatar, desarrollado por el grupo de seguridad y sistemas de Eurecom. Del mismo modo que este, se trata de un marco de orquestación de código abierto, que permite la interoperación de distintos marcos de análisis binario, emuladores, depuradores y dispositivos físicos. Realiza una generalización de Avatar, permitiendo la orquestación a un número arbitrario de herramientas diferentes.

BitBlaze

Es una plataforma de análisis binario, escrita en C, C++ y OCaml, promovida por la Universidad de Berkeley [3]. Dispone de una arquitectura extensible dividida en tres componentes: *Vine*, un analizador estático, TEMU, el componente de análisis dinámico, y *Rudder*, un componente de análisis concólico y simbólico que combina los dos anteriores. *Vine* convierte el código binario en un lenguaje intermedio y proporciona varias herramientas de análisis estático (flujo de control y datos, optimización y cálculo de la precondición más débil). El emulador TEMU proporciona un entorno de ejecución dinámico. Este emulador, construido sobre la base de QEMU, proporciona un API que permite desarrollar plugins donde implementar un análisis a medida. Finalmente, *Rudder* se apoya en los otros dos componentes para gestionar la ejecución simbólica y concreta. Además, identifica los predicados de las rutas necesarios e interroga a los solucionadores que permiten determinar si la ruta es posible.

Capítulo 3

Solucionadores SMT

Los trabajos iniciales sobre SMT aparecieron a finales de los años 70 del pasado siglo, pero es a partir de finales de los 90 cuando surgen los solucionadores SMT actuales. Un área que experimenta a partir de ese momento un gran desarrollo, encontrando aplicaciones prácticas en la verificación de sistemas, modelos y la generación de casos de test.

3.1. Descripción general

Se conoce como **Satisfiability Modulo Theories (SMT)** al problema de verificar la satisfacción de fórmulas lógicas de primer orden (FOL - "*First Order Logic*") sin cuantificadores, referidas a una o más teorías de primer orden (T), como puede ser la teoría de los números reales, los números enteros o las teorías respecto a tipos de estructuras (listas, *arrays* o vectores de bits). Los modernos solucionadores siguen el paradigma DPLL(T) ("*Davis-Putman-Logemann-Loveland modulo Theories*") que marca una implementación separada para cada teoría, por lo que los distintos solucionadores SMT no tienen por qué implementar la misma teoría. La descripción rigurosa de dichas teorías se conoce con el nombre de SMT-LIB (SMT Library), que sigue la notación de las "*S-expressions*" del lenguaje Lisp.

Los *solucionadores SMT*, son una generalización del problema "*Boolean Satisfiability Problem*" (SAT) donde las variables booleanas son reemplazadas por predicados de acuerdo a la teoría subyacente (T), ver figura 3.1.

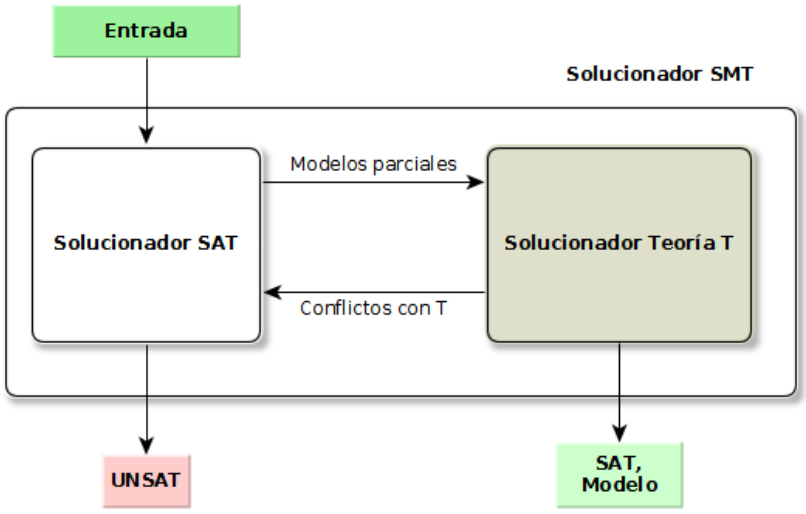


Figura 3.1: Modelo conceptual de un solucionador SMT

De forma general, puede definirse un **solucionador SMT** como la integración de un solucionador SAT y un solucionador para una determinada teoría (T). La entrada al solucionador SMT es una consulta que viene dada por un conjunto de condiciones expresadas de acuerdo a la sintaxis de la teoría T. El resultado de salida de este sistema puede ser la no satisfacción de la expresión de entrada (UNSAT) o la satisfacción de la misma (SAT), en cuyo caso se ha debido encontrar un modelo que la satisfaga. Esto es, el solucionador SMT incluye un procedimiento de decisión que viene implementado por el solucionador de la teoría T.

Aunque SMT puede ser aplicado a cualquier tipo de teoría, solo unas pocas son interesantes desde el punto de vista del análisis y verificación de programas. Entre estas encontramos:

- **Teoría de la igualdad (QF_UF)**, que no restringe los valores posibles de los símbolos empleados. A veces llamada teoría de la igualdad con funciones no interpretadas ("*equality with uninterpreted functions*" - EUF). La teoría permite operaciones lógicas, igualdades y desigualdades.

- **Teorías de aritmética lineal** no cuantificada sobre R y Z (QF_LIR, QF_LIA). Se soportan las operaciones '+', '−', '·'(solo la forma $c \cdot x$, donde c es un coeficiente y x una variable) para los enteros, la operación ':'para los reales y

los símbolos para igual ('=') o desigualdad ('>', '<', ...). Muchos solucionadores utilizan el algoritmo del *simplex* para la toma de una decisión.

- **Teorías sobre lógica de diferencias** sobre R y Z (QF_IDL, QF_RDL). Se cubre la aritmética lineal correspondiente a las inecuaciones, expresiones de la forma $x - y \leq c$.

- **Teoría de los bit-vectores de tamaño fijo (QF_BV)**, donde cada variable representa un vector de bits de longitud finita. Además de las operaciones aritméticas estándar se permiten las operaciones lógicas (NOT, AND, OR, XOR) y los desplazamientos de bits.

- **Teoría de *arrays*** sin extensión (QF_AX) y con ella (QF_A). Define el uso de los *arrays*, que tienen dos operaciones especiales: la escritura y la lectura de una posición del *array*. Se trata de una teoría vaga que permite diferentes extensiones.

3.2. Algunos solucionadores SMT

A continuación, se describen brevemente algunos de los solucionadores SMT más frecuentemente utilizados, un estudio comparativo puede encontrarse en [10]:

- **STP ("*Simple Theorem Prover*")**: Se trata de un solucionador para la teoría de los vectores de bits, sin cuantificadores (QF_BV). Su lenguaje de entrada posee la mayoría de las funciones y predicados implementados en un lenguaje de programación como C, con la excepción de la aritmética y los tipos de coma flotante, que no son soportados. De esta forma, STP implementa todas las operaciones aritméticas, las operaciones lógicas sobre los bits, las relacionales, los multiplexores y la concatenación y extracción de bits. Todas las operaciones sobre vectores de bits son trasladadas a operaciones sobre bits. STP convierte un problema de decisión en su lógica proposicional CNF ("*Conjunctive Normal Form*") antes de llamar a MiniSAT, el solucionador SAT subyacente.

- **Z3**: Este es un solucionador desarrollado por Microsoft e implementado en C++, publicado bajo la licencia "*Microsoft Research License Agreement*" (MSR-

LA), está disponible para C, C++, .NET, Python, Java y OCaml. Soporta una extensión de SMT-LIB v2.0. Considerado como uno de los mejores solucionadores SMT, soporta todas las teorías de la SMT-LIB. Es capaz de generar pruebas para las fórmulas no satisfechas y extraer de la expresión de entrada cuáles son las clausulas que no satisfacen la teoría subyacente.

- **CVC4 ("*Cooperating Validy Checker*")**: Es un proyecto de código abierto de la Universidad de Nueva York junto con la Universidad de Iowa. Está desarrollado en C++ y se publica bajo la licencia BSD ("*Berkeley Software Distribution*"). Soporta todas las teorías de SMT-LIB. Dispone de un formato propio nativo (lenguaje CVC), aunque también soporta SMT-LIB v2.0.

- **Yices 2**: Desarrollado en C por "*Standford Research Institute*" (SRI International) y distribuido bajo la licencia Yices. Implementa la lógica SMT-LIB, sin cuantificadores, para funciones no interpretadas, aritmética lineal y no lineal, lógica de diferencias para enteros y reales, *arrays* y vectores de bits.

- **VeriT**: Creado por la Universidad de Nancy, el "*Institut National de Recherche en Informatique et Automatique*" (INRIA) y la Universidad de Rio Grande do Norte. Está escrito en C, es un proyecto de código abierto y se distribuye bajo la licencia BSD. Proporciona soporte para las lógicas sin cuantificadores SMT-LIB e integra cierto grado de razonamiento de acuerdo a los cuantificadores.

- **MathSAT 5**: Proyecto de la "*Fondazione Bruno Kessler*" y la Universidad de Trento. Escrito en C++ está disponible para investigación académica y evaluación. Soporta SMT-LIB 2.0 y 1.2 e implementa la mayoría de las teorías subyecentes. Su API de programación permite la integración de un solucionador SAT externo.

- **SMTInterpol**: Desarrollado por la Universidad de Freiburg, está escrito en Java y disponible bajo la licencia "*GNU Lesser General Public License*" (LGPL). Acepta SMT-LIB 1.2 y 2.0 como formatos de entrada y soporta las teorás de las funciones no interpretadas, aritmética lineal sobre enteros y reales, y la combinación de estas teoraías.

3.3. Aplicaciones

Uno de los principales objetivos de SMT es la creación de herramientas de verificación con un alto grado de abstracción, pero al mismo tiempo veloces y de fácil automatización. De hecho, desde un punto de vista simplificado, un sistema de ejecución simbólica puede considerarse como la integración de un ejecutor *booleano*, que ejecute las instrucciones, y un solucionador SMT, que verifique las satisfacción de las condiciones de ruta.

Por otra parte, los solucionadores SMT constituyen módulos independientes de software que pueden ser integrados en estas herramientas a través del API que exponen. También pueden ser usados desde la línea de commandos. Por ejemplo, el listado 3.1 muestra una consulta sencilla en SMT-LIB de la operación *XOR*.

Listado 3.1: Operación XOR, "*xor.smt2*"

```
(set-logic QF_BV)
(set-info :smt-lib-version 2.0)
(set-info :category "check")
(set-info :status sat)
(declare-fun v0 () Bool )
(declare-fun v1 () Bool )

(assert (xor v0 v1))

(check-sat)
(get-value (v0 v1))
(exit)
```

Una invocación del solucionador SMT, en este caso STP, resuelve la expresión como satisfacible, dando un modelo consistente con la expresión:

```
$ stp xor.smt2
sat
(
( |v0| true )
( |v1| false )
)
```

Sin embargo, un ejemplo de la teoría de aritmética lineal (QF_LIA), como el que

se muestra en el listado 3.2 ($x + 2y = 20; x - y = 2$), no puede ser resuelto por STP.

Listado 3.2: Sistema de ecuaciones, "*equations.smt2*"

```
(set-option :produce-models true)
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(assert (= (+ x (* 2 y)) 20))
(assert (= (- x y) 2))
(check-sat)
(get-value (x y))
(exit)
```

En este caso, es necesario utilizar un solucionador que soporte QF_LIA, como es el caso de CVC4. Y así, al emplear CVC4 el resultado es SAT, junto con el modelo encontrado:

```
$ cvc4 equations.smt2
sat
((x 8) (y 6))
```

3.4. Limitaciones

Los solucionadores SMT han demostrado ser herramientas muy potentes, pero que no están exentos de limitaciones para su aplicación. Entre esas limitaciones se puede encontrar:

1. Alto consumo de recursos de memoria: El empleo de solucionadores SMT en herramientas automáticas suele generar fórmulas de costosa solución. Y en el caso de la no satisfacción, se debe calcular el mínimo subconjunto que lo prueba, pudiendo alcanzar un tamaño considerable. En ocasiones una solución particular a un problema específico es más eficiente.

2. Lentitud: por los mismos motivos que el punto anterior, el empleo de solucionadores SMT va a ralentizar el funcionamiento de cualquier herramienta que los utilice.

3. Dificultad de las teorías: A menudo, los programas utilizan teorías, expresiones o funciones, que están fuera de las posibilidades de los solucionadores SMT. Caso de la aritmética de punteros, ciertas expresiones aritméticas no lineales (ej.: $x^3 = y$) o el uso de funciones criptográficas o de "*hashing*" (ej.: $x = hash(y)$. Téngase en cuenta que en este último caso, dichas funciones están especiálmente pesadas para evitar el tipo de razonamiento que implementan los solucionadores SMT.

Capítulo 4

Ejecución simbólica

La ejecución simbólica es una de las técnicas que pueden emplearse para automatizar el testeo de software mediante la generación de casos de test y alcanzar así una gran cobertura de código. Fue introducida en 1976 por **James King** del *IBM Thomas J. Watson Research Center* [11] con su herramienta EFFIGY. No se trata, por tanto, de una técnica novedosa. Sin embargo, desde 2005 está conociendo un fuerte resurgimiento como área de investigación debido a la mejora del *hardware* y a los avances conseguidos en las herramientas de comprobación automática de teoremas.

En este capítulo se realiza una exposición de la ejecución simbólica, sus principales variantes y se indican las principales herramientas basadas en ella.

4.1. Ejecución simbólica clásica

La **ejecución simbólica** es una técnica de análisis y testeo de programas que realiza la ejecución de un programa dentro de un dominio simbólico donde las entradas al programa son valores simbólicos que representan cualquier valor, sin tomar uno en concreto. Puede verse como una mejora en las técnicas de test, puesto que en lugar de realizar un conjunto de test específicos, se ejecutan conjuntos de clases de test, por lo que el resultado equivale a un número elevado de casos concretos [11].

4.1.1. Descripción

La ejecución simbólica trata una variable de entrada como un valor simbólico, que representa un rango de valores. Se busca analizar el comportamiento del programa de acuerdo a cada combinación de valores simbólicos, consiguiendo encontrar de forma sistemática rutas posibles de ejecución del programa y establecer las condiciones bajo las que se ejecutan.

La asignación de variables queda de esta forma extendida, pues una variable puede tener un valor concreto o una *expresión simbólica*. Una **expresión simbólica** se define como una expresión que representa una función sobre las variables simbólicas y que será generada conforme se evalúan las instrucciones del programa. La evaluación de una expresión simbólica expande las variables incluidas en la expresión y transforma la instrucción en una operación aritmético-lógica.

De manera formal [12], una **ruta de ejecución** (π) de un programa secuencial determinista (P) se puede definir como:

$$\pi = l_0 \xrightarrow{instr_1} l_1 \xrightarrow{instr_2} l_2 ... \xrightarrow{instr_n} l_n \tag{4.1}$$

donde, l_i son las correspondientes localizaciones de control por las que va pasando la ejecución, y la instrucción ($instr_i$) puede reducirse a las siguientes, tomadas como modelo:

- La asignación a una variable (v) de: una expresión aritmética, una constante, o la aplicación de un operador.
- La instrucción "**skip**" o de no operación.
- La instrucción "**halt**", indicando un final normal de la ejecución.
- La instrucción "**abort**", indicando un final anormal de la ejecución del programa. La existencia de una vulnerabilidad puede modelarse como un caso especial de "**abort**".
- Una rama $if(c)$ donde "c" es una expresión condicional, y la rama alternativa ("*else*") viene dada por $if(\neg c)$. El caso del bucle "**while**" se reduce a un caso

particular de este.

Una ruta de ejecución (π) es posible si existe una entrada al programa que la cubra, de entre el conjunto posible de entradas o de test (T). Así, la ejecución concreta se puede representar con (τ, π), mientras que la ejecución simbólica queda expresada por $(*, \pi)$, puesto que la entrada de test es arbitraria. Puede verse que la ejecución simbólica es una generalización de la ejecución concreta tradicional, y podemos concluir que si el programa no posee un árbol de ejecución infinito, resulta ser un método completo.

Se observa, también, que la semántica de ejecución de las instrucciones queda extendida debido a que necesita manejar el tratamiento de *expresiones simbólicas*, tanto en las entradas, como en la salida del programa. Un ejemplo de esto es la instrucción "$if(c)$" que de acuerdo a la condición ("c") puede evaluarse a: verdadero, falso o indefinido. En el caso de que la condición no resulte con valor indefinido se exploran las instrucciones restantes mediante una bifurcacin en el proceso, asumiendo el valor "*verdadero*" para uno de los casos y "*falso*" para el otro.

Por otro lado, el **estado del programa** en cada localizacin de control (l_i) se extiende para mantener un conjunto de condiciones con la asunción de un valor durante la ejecución del proceso. Este conjunto de condiciones recibe el nombre de **condición de ruta** ("*path condition*"), denotado como "*pc*". Así, en cada localización de control (l_i) existe una condición de ruta (pc_i), y cuando se produce un evento, $l_i \xrightarrow{instr} l_{i+1}$, si "$instr$" es $if(c)$, entonces la nueva condición de ruta es $pc_{i+1} = pc_i \wedge c$, en caso contrario, $pc_{i+1} = pc_i$.

Con la extensión anterior, el **estado del programa** pasa a quedar definido por la tupla $\langle pc, l, mem \rangle$, donde pc es la condición de ruta, l es la localización y mem es el mapa de memoria simbólico, esto es, la relación entre la variable simbólica y la expresión que representa sus posibles valores.

La condición de ruta (pc), que como hemos visto es una expresión lógica sobre las variables simbólicas, necesita ser evaluada durante la ejecución para verificar la posibilidad de su satisfacción, y en caso de no poder ser satisfecha la ruta queda descartada. Es aquí donde entran en juego y resultan críticos los solucionadores SMT. Además, una vez terminado el programa, la condición de ruta resultante pasa nuevamente al solucionador para conseguir el caso de test que la hace posible.

De acuerdo con lo anterior, el proceso que describe la **ejecución simbólica clásica** queda descrito de manera global por el pseudocódigo del algoritmo 1.

Algoritmo 1: Ejecución Simbólica Clásica

```
1:  estado_0 ← ⟨true, l_init, mem_init⟩
2:  almacenar(estado_0)
3:  while ( ∃ estados almacenados )
4:      ⟨pc, l, mem⟩ ← recuperar_estado();
5:      if ( pc se satisface |mem )
6:        for each ( evento l --instr--> l' )
7:          if ( instr es abort )
8:              τ ← solve(pc, mem);
9:              T := T ∪ {τ};
10:         else if ( instr es halt );
11:             τ ← solve(pc, mem);
12:             T := T ∪ {τ};
13:         else if ( instr es if(c));
14:             estado_siguiente ← ⟨pc ∧ c, l', mem⟩;
15:             almacenar_estado(estado_siguiente);
16:         else if ( instr es v := exp);
17:             estado_siguiente ← ⟨pc, l', mem[v ← exp]⟩;
18:             almacenar(estado_siguiente);
19:         end if;
20:       end for;
21:     end if;
22: end while;
23: return T;
```

Según se aprecia en el algoritmo 1, las fases del procedimiento de ejecución

simbólica sobre un programa son las siguientes:

- **Inicialización**: Se definen las variables simbólicas y se inicia la ejecución. Esta crea el estado inicial y lo almacena en una lista de trabajo. El estado inicial queda representado por la tupla $\langle true, l_{init}, mem_{init} \rangle$, donde la condición de ruta toma el valor "*true*".

- **Selección del estado**, de la lista de trabajo, y verificación de que se trata de un estado válido, es decir, que la condición de ruta se satisface de acuerdo con el estado de la memoria.

- **Ejecuciones secuenciales** provocadas por un evento del tipo $l \xrightarrow{v:=exp} l'$. En este caso se calcula un nuevo estado de la memoria ($mem[v \leftarrow exp]$) asignando la expresión exp a la variable v. Las ejecuciones secuenciales se realizan hasta que se encuentra una condición "$if(c)$" o la salida del programa.

- **Ejecución de una instrucción "$if(c)$"**: Se evalúa la condición de acuerdo con la condición de ruta, que determina la posibilidad de la rama representada por $if(c)$, y la rama alternativa $if(\neg c)$. Las ramas que son posibles son exploradas, produciéndose una bifurcación en el árbol de ejecución.

- **Detección de un error en la ejecución**: suceso del evento "*abort*" ($l \xrightarrow{abort} l'$), que termina el proceso sobre la rama explorada.

- **Finalización**, evento $l \xrightarrow{halt} l'$: Se obtiene la condición de ruta completa y se genera el caso de test. Si hay más estados en la lista de trabajo, se continúa con ellos.

Como puede observarse, el procedimiento de ejecución simbólica continúa hasta que todos los estados han sido tratados, con la excepción de la existencia de rutas de ejecución sin final o la posible generación de infinitas rutas de ejecución.

El conjunto de rutas ejecutadas durante la ejecución simbólica constituye el llamado **árbol de ejecuciones simbólicas** y representa todas las rutas posibles en el programa. Cada nodo del árbol indica una bifurcación causada por la instrucción "$if(c)$", y cada hoja corresponde a una terminación del procedimiento de ejecución simbólica, con la condición de ruta asociada, que es única para cada hoja. Además,

cada caso de test corresponde con una y solo una condición de ruta y existe al menos un caso de test para cada ruta posible.

Por otra parte, la ejecución simbólica no especifica la forma de almacenar o seleccionar los estados. Por tanto, entre las estrategias que se puede encontrar en los distintos motores de ejecución simbólica, a veces combinando varias, se pueden destacar las siguientes [13]:

- **Búsqueda "*Depth-first*"**: Es la estrategia más sencilla, donde la lista de estados se implementa como un "*stack*", seleccionándose el estado creado más recientemente. De esta forma, siempre se completa una ruta de ejecución antes de comenzar con la siguiente. En casos donde el árbol de ejecución está fuertemente desequilibrado esta metodología puede implicar un alto esfuerzo en la exploración para una pequeña parte de dicho árbol.

- **Búsqueda "*Breath-first*"**: En este caso, la lista de estados es de tipo FIFO ("*First-In, First-Out*"), siendo seleccionado el estado más antiguo. Da la misma oportunidad de exploración a las distintas ramas, pero tiene el inconveniente de que si el punto de interés tiene una profundidad elevada en el árbol de ejecución este método puede tardar un tiempo considerable en llegar.

- **Aleatorio**: Los estados se seleccionan aleatoriamente con igual probabilidad. Resultan favorecidos los estados con poca profundidad en el árbol de ejecución. En contrapartida, y dada la aleatoriedad los resultados pueden ser difíciles de reproducir.

- **Optimización de cobertura de código**: Se busca elegir estados que impliquen mayor cobertura de código. Normalmente se asigna un peso al estado dependiendo de su distancia al código no cubierto. El motor de ejecución simbólica selecciona el estado aleatoriamente de acuerdo al peso asignado. Este método incrementa la complejidad pero consigue una mayor eficiencia en la cobertura de código.

4.1.2. Limitaciones

Aunque la ejecuciíon simbólica es una técnica potente para el razonamiento sobre los estados del programa y la búsqueda de fallos de software ("*bug*"), también cuenta con algunas limitaciones para su aplicación práctica:

1. Su escalabilidad está seriamente condicionada por el **problema de explosión de rutas**. En el mejor de los casos, se estima que el número de rutas crece exponencialmente en proporción al número de sentencias condicionales. En el peor, este número será infinito si el programa presenta bucles infinitos. La explosión de rutas conlleva una restricción computacional, debido a que todas las rutas deben ser exploradas, y otra debida a los límites de memoria.

2. Esta limitada por el coste de las consultas realizadas sobre los solucionadores de restricciones ("***SMT solvers***"). En general, la solución para las consultas generadas por la ejecución simbólica para la teoría de los vectores de bits (QF_BV) es un problema NP-completo, y en muchos casos no se pueden resolver en un tiempo razonable. También con los casos de complejidad elevada en las restricciones (por ejemplo, restricciones no lineales) o de llamadas a funciones de las que no se dispone del código fuente, como la interacción con el entorno (sistema operativo, red o usuario).

3. El coste computacional elevado de la ejecución simbólica, que es de varios órdenes de magnitud mayor que el de la ejecución nativa, y que puede ser fuertemente limitante en el análisis de ciertos tipos de programas.

4. Se trata de una técnica de caja blanca ("*white-box*") donde se necesita el código fuente del programa (o su equivalente) y el uso de librerías que modelen ciertos comportamientos.

5. Presenta un elevado número de falsos positivos, al evaluarse la condición de ruta al final.

4.2. Ejecución concólica

Para afrontar los problemas de la ejecución simbólica como la explosión de rutas, la imposibilidad de manejar restricciones no resolubles o la dificultad de representar simbólicamente estructuras complejas de datos, aparece la ejecución concólica ("*concolic execution*"). Se trata de una estrategia que combina la ejecucin concreta y la simbólica (**Concolic** = **Conc**rete + Symb**olic**), y es utilizada por muchas de las herramientas de ejecución simbólica actuales.

La **ejecución concólica** puede sintetizarse en las siguientes ideas centrales:

1. Ejecución inicial del programa con una entrada aleatoria sobre una o varias de las entradas de test. Estas entradas definen una condición inicial durante la ejecución, que se añade a las condiciones de la rama.

2. Después de completar la ejecución, se niega una de las condiciones en la ruta y se pasa al solucionador SMT para conseguir un modelo, o para ver que las condiciones de ruta no son posibles.

3. Se lanza una nueva ejecución con la generación de un nuevo caso de test a partir de los nuevos valores concretos encontrados, forzando la exploración de una nueva ruta. Con las sucesivas ejecuciones, el árbol de ejecución va quedando restringido.

Puede verse que la ejecución concólica es guiada por las ejecuciones concretas que van generándose. Además, en este caso, junto al estado simbólico hace falta mantener un estado concreto que guarde el mapeo de las variables a su valor concreto.

Es necesario resaltar que la ejecución concólica puede puentear el problema de una restricción no resoluble, pero no garantiza la completitud en la exploración (existirán rutas que no se exploran) y tampoco existe una garantía de que todas las restricciones generadas sean resolubles.

Como características importantes de la **ejecución concólica**, podemos destacar:

- Permite remplazar con facilidad las variables simbólicas por valores concretos que satisfagan las condiciones de ruta.

- Puede realizar llamadas directas al sistema, aunque se pierde el carácter simbólico en dichas llamadas.

- Permite tratar casos demasiado complejos para los solucionadores SMT.

4.3. Ejecución generada

La ejecución generada (EGT - "*Execution-generated Testing*") es también una mezcla de ejecución simbólica y concreta. A diferencia de la ejecución concólica, EGT trabaja en el dominio simbólico, pero cambia a la ejecución concreta si todos los datos de la operación son concretos o existe una gran dificultad para resolver la restricción. En consecuencia, los valores concretos no se inicializan inmediatamente, sino cuando estos son necesarios.

Cabe destacar, que la ejecución concólica y la EGT tienen la ventaja de no generar falsos positivos, ya que aseguran que las rutas obtenidas para el caso de error son posibles y que el test generado cumplirá con la ruta y el error se producirá.

4.4. Otras variantes

La técnica de la ejecución simbólica presenta numerosas variantes dada la facilidad de la extensión de su algoritmo básico. Entre ellas podemos encontrar la ejecución simbólica generalizada (GSE - "*Generalized Symbolic Execution*") con el manejo de programas multihilo y estructuras de datos recursivas como entradas. También la ejecución concólica ha sido extendida para la verificación de programas con concurrencia, aplicando metodologías adicionales para la reducción del número de rutas exploradas.

4.5. Implementación

La implementación de la de ejecución simbólica para el análisis de programas puede realizarse principalmente de la formas siguientes:

- **Transformación**: Se trata de transformar un programa en otro que opere con valores simbólicos. Es una una solución portable y adecuada a las tecnologías Java y .NET. Sin embargo, es difícil de implementar.
- **Instrumentación**: En este caso, se insertan retrollamadas ("*callback hooks*") de manera que la ejecución simbólica se ejecuta en un segundo plano durante la ejecucin normal del programa. Técnica implementada por KLEE o CUTE y adecuada para programas escritos en lenguaje C.
- **Máquina virtual simbólica**: Aplicable a Java y .NET, es flexible pero difícil de implementar y no portable.

4.6. Principales herramientas

En la actualidad son muchas las herramientas que hacen uso de la ejecución simbólica, o alguna de sus variantes. Entre las más destacadas podemos citar las siguientes [14]:

- **JPF-SE y Java PathFinder**: JPF es una extensión de Java PathFinder (desarrollado por la NASA) siendo un verificador de modelos de propósito general. Se trata de una herramienta de código abierto desde 2003.
- **DART ("*Directed Automated Random Testing*")**: Mezcla EGT con testeo aleatorio y técnicas de verificación de programas con el fin de revisar todas las rutas de ejecución posibles, al mismo tiempo que se verifican varios tipos de errores en cada ejecución. Consigue aliviar las imprecisiones de la ejecución simbólica mediante valores concretos y aleatorios, a costa de realizar una ejecución simbólica parcial. Implementado en Bell Labs para el testeo de programas en C.
- **CUTE/jCUTE**: CUTE ("*Concolic Unit Testing Engine*") y jCUTE (CUTE para Java) extiende DART para manejar programas multihilo que manipulan estructuras dinámicas. Evita las imprecisiones debidas al análisis de punteros aproximando la resolución de restricciones sobre los punteros. Ambas herramientas fueron desarrolladas en la Universidad de Illinois.

- **CREST**: Una herramienta de código abierto para el testeo de programas en C. Funciona insertando código de instrumentación en el programa para realizar la ejecución simbólica.

- **SAGE (“*Scalable Automated Guided Execution*”)**: Desarrollado por Microsoft permite la automatización como de caja blanca de test borrosos (“*Whitebox Fuzzing*”). Utiliza la generación sistemática de test dinámicos y los amplía desde el test unitario al test de toda la aplicación. Se trata de una herramienta concólica, que a partir de test aleatorios lanza la ejecución simbólica para generar las restricciones de las rutas encontradas. Después, se niegan las restricciones y se resuelven para generar nuevas entradas de test. En las comprobaciones durante la exploración de rutas se utilizan herramientas verificadores como Purify, Valgrind o AppVerifier.

- **Pex**: Desarrollado por Microsoft, implementa ejecución simbólica dinámica para generar casos de test para .NET y lenguajes C#, VisualBasic y F#. Pex modeliza simbólicamente casi todas las instrucciones de la plataforma .NET. Utiliza Z3 como solucionador de restricciones usando aproximaciones para los casos no soportados por Z3 (teoría de “*strings*” y de aritmética flotante). Soporta la generación de entradas de test para tipos tanto primitivos como complejos, y combina varias estrategias de búsqueda. Además Pex viene con un framework que facilita la reutilización de modelos para las librerías .NET.

- **EXE (“*Execution Generated Executions*”)**: Desarrollado en la Universidad de Standford, es un motor de ejecución simbólica para programas escritos en C, dando especial importancia al código de sistemas, para lo que EXE modela la memoria con una precisión a nivel de bit. Para analizar un programa es necesario incluir marcas de las localizaciones de memoria que serán tratadas como simbólicas [15]. Después, la compilación se realiza a través del compilador de EXE (“*exe-cc*”) que realiza una instrumentación del código usando CIL (“*C Intermediate Language*”), antes de compilar de forma estándar (usando “*gcc*” por ejemplo). Utiliza el solucionador STP para implementar el procedimiento de decisión para vectores de bits y *arrays*.

- **KLEE**: Se trata de un rediseño de EXE sobre la infrastructura de compilación

LLVM. Realiza una mezcla de ejecución concreta y simbólica y emplea métodos heurísticos de exploración de rutas con el fin de ampliar la cobertura de código. Consigue una representación más compacta de los estados que su predecesor y es destacable también la mejora de la modelización del entorno, consiguiendo una interaccón sencilla. En junio de 2009 fue liberada como código abierto y desde entonces viene siendo extendida por varios grupos de investigación.

- **S2E ("*Selective Symbolic Execution*")**: Se trata de una plataforma para el modelado rápido de análisis a medida. Consiste en una máquina virtual adaptada a partir de QEMU (un emulador de procesadores que soporta ARM, x86 y SPARC entre otros), un traductor binário dinámico (DBT - "*Dynamic Binary Translator*") y un motor embebido de ejecución simbólica (basado en KLEE). Es capaz de analizar directamente el código binario x86, x86-64 y ARM, para lo que realiza una conversión a LLVM [16].

Capítulo 5

KLEE

En este capítulo se presenta el motor de ejecución simbólica KLEE, ampliamente utilizado en la actualizad por la comunidad investigadora. Antes de pasar a su descripción se hace una rápida presentación de LLVM, la base sobre la que se sustenta KLEE. Después, se abordan las posibilidades que ofrece KLEE como intérprete independiente y su capacidad para integrarse en otros proyectos. Finalmente se indica también algunas de sus limitaciones.

5.1. El proyecto LLVM

El **proyecto LLVM** es una colección modular y reutilizable de tecnologías para el desarrollo de compiladores escrito en C++, creado en la Universidad de Illinois. El nombre LLVM inicialmente provenía de "*Low Level Virtual Machine*", pero con las sucesivas ampliaciones del proyecto se abandonó, quedando solo LLVM. Actualmente es un proyecto de código abierto bajo la licencia UIUC ("*University of Illinois/NCSA Open Source License*"), una licencia compatible con la licencia GPL ("*GNU General Public License*").

Básicamente LLVM es una librería para construir compiladores, ya sea su parte "*front-end*" (parser y lexer) o "*back-end*" (transformaciones de código intermedio a código máquina), y también software orientado a lenguajes. Al contrario que el clásico "*gcc*" está orientado a la modularidad y a la reutilización con el fin de facilitar la escritura de distintos compiladores. Sus principales subproyectos son:

- LLVM IR ("*LLVM Intermediate Representation*"): El lenguaje intermedio de representación del compilador. Se trata de un lenguaje ensamblador genérico. Por otro lado, LLVM también proporciona soporte para compilación JIT ("*Just In Time*") de este código intermedio.

- Núcleo de LLVM: Compuesto por librerías que implementan la optimización a nivel de código intermedio.

- CLang: Es el compilador nativo de LLVM para C, C++ y Objective-C. Viene acompañado por un analizador de código estático ("*Clang Static Analyzer*").

- Librerías LLVM: Constituyen la parte reutilizable de la infraestructura LLVM. Entre estas se encuentran la libc++, una implementación de la librería C++ estándar con soporte para C++11, y la libc++abi, una implementación con soporte de bajo nivel (ABI - "*Application Binary Interface*") como puede ser la gestión de memoria o el manejo de excepciones.

- LLDB: Un depurador nativo que se apoya fuertemente sobre el parser de CLang y el desensamblador LLVM.

- Soporte para generación de código ("*compiler-rt*") en distintas arquitecturas (i386, X86-64, SPARC64, ARM, PowerPC, PowerPC 64) y sistemas operativos (AuroraUX, DragonFly BSD, FreeBSD, NetBSD, Linux, Darwin).

La orientación agnósitica respecto al lenguaje de LLVM ha permito su empleo en la construcción de compiladores para un número elevado de lenguajes, entre los que se encuentran: Ada, ActionScript, Common Lisp, C#, Frontran, Haskell, Lua o Ruby, entre otros.

5.2. Descripción y arquitectura

KLEE es una herramienta de ejecución simbólica diseñada en la Universidad de Standford, por el grupo de Dawson Engler, para trabajar con código fuente en lenguaje C. Se trata, al igual que LLVM, de un proyecto de código abierto bajo la licencia UIUC "*University of Illinois/NCSA Open Source License*".

Como se describe en [17], básicamente KLEE funciona como un híbrido entre un sistema operativo para procesos simbólicos y un intérprete de bitcode o ensamblador LLVM, en concreto LLVM IR. En el diseño de KLEE se han tenido en cuenta los problemas de escalabilidad que afronta todo motor de ejecución simbólica, esto es:

- El crecimiento exponencial del número de rutas de la ejecución simbólica.
- El alto coste de los solucionadores SMT.
- El problema de la interacción con el entorno.

La figura 5.1 muestra la arquitectura general de un entorno de ejecución simbólica basado en KLEE. El código fuente inicial (escrito en C) es compilado en bitcode, utilizando el compilador "*clang*" o "*llvm-gcc*" (herramientas incluidas en LLVM), antes de pasar a KLEE y ser interpretado. El núcleo de KLEE es un intérprete que mantiene la información necesaria para cada uno de los estados activos (cada estado, tiene registro de archivo, *stack*, *heap*, contador de programa y condición de ruta). Como motor simbólico, KLEE toma secuencialmente un estado y ejecuta de forma simbólica una instrucción en el contexto del estado. El proceso continúa hasta que no hay más estados por seleccionar o se ha alcanzado un tiempo límite definido por el usuario.

En caso de llegar a una condición de rama ($if(c)$), KLEE interroga al solucionador SMT (STP, Z3 o minisat) sobre la certeza de la condición para poder continuar. En el caso de que se den las dos posibilidades, KLEE clona el estado y lanza un nuevo proceso hijo para poder definir las dos rutas resultantes.

Las operaciones potencialmente peligrosas (como divisiones, carga o almacenamiento en memoria) generan implícitamente ramas a partir de la posibilidad de que alguna entrada pueda causar un error (por ejemplo, división por cero). En el caso de detectar un error, se genera el caso de test para llegar hasta él y se continúa con la exploración añadiendo la condición para evitarlo como una restricción sobre la ruta. Si no ha habido error, KLEE genera los casos de test al finalizar la exploración de las rutas posibles.

La etapa de consulta que se hace sobre el solucionador SMT es la más costosa en términos de tiempo computacional. Es por esto que KLEE necesita implementar

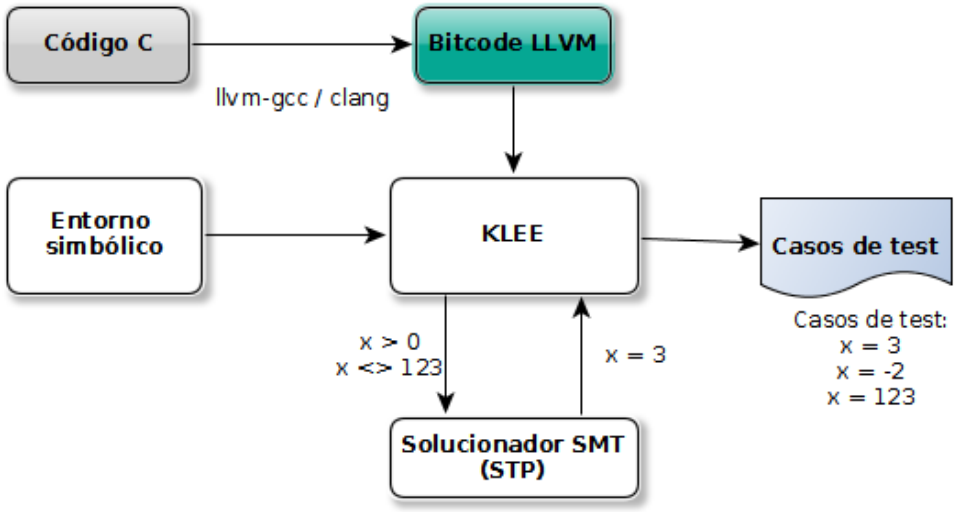


Figura 5.1: Arquitectura general de KLEE

técnicas de simplificación en las consultas realizadas con el fin de paliar este problema. Los principales métodos de simplificación aplicados son:

1. **Reescritura de expresiones**, usando las misma metodología que los compiladores. Casos de estos métodos son: simplificación aritmética (ej.: $x + 0 = x$), simplificación lineal (ej.: $2*x-x = x$) o reducción de la longitud de la expresión (ej.: $x * 2n = x << n$).

2. **Simplificación del conjunto de restricciones**: Durante la ejecución simbólica pueden aparecer gran número de restricciones. KLEE es capaz de reescribir los conjuntos de restricciones que se aplican sobre las mismas variables.

3. **Concretización implícita**: Se trata de localizar y simplificar las restricciones que implican una condición concreta sobre una variable (ej.: $x+1 = 10 \rightarrow x = 9$).

4. **Independencia de restricciones**: En muchas ocasiones un conjunto de restricciones queda dividido en grupos disjuntos según las variables implicadas. En esta situación KLEE puede eliminar las restricciones que no tienen efecto sobre la consultas de interés.

5. **Caché de contraejemplos**: Frecuentemente las consultas son recurrentes y pueden reducirse empleando una caché que mapee conjuntos de restricciones sobre contraejemplos. Si la caché se implementa teniendo en cuenta los subconjuntos y superconjuntos de restricciones, se puede tener nuevas vías para eliminar consultas. Las reglas aplicadas son las siguientes:

   a) Si un subconjunto de restricciones no tiene solución, el conjunto del que procede tampoco.

   b) Si un superconjunto de restricciones tiene solución, cualquier subconjunto de este también la tiene.

   c) Cuando un subconjunto de restricciones tiene solución, es probable que también la tenga el conjunto al que pertenece. Dado que es menos costoso la comprobación de soluciones, KLEE comprueba las soluciones de todos los subconjuntos de acuerdo a las restricciones del conjunto origen, y poder así encontrar una solución a dicho conjunto.

Uno de los temas a tener en cuenta en la ejecución simbólica es la **interacción con el entorno**, como puede ser los argumentos pasados por línea de comandos, los archivos, las variables de entorno, etc. Nótese que ahora operaciones como la lectura de un archivo necesita saber si se trata de un archivo concreto o uno simbólico. Esta modelización puede hacerse de dos formas: a nivel de llamadas al sistema (ej.: "*open()*") o a nivel de la librería C estándar (ej.: "*fopen()*"). En el caso de KLEE, debido a la gran cantidad de funciones de la librería estándar de C, se ha optado por modelizar las llamadas al sistema y utilizar una de las implementaciones de la librería estándar de C, **uClibc**, mucho más pequeña y que inicialmente fue escrita para sistemas con Linux embebido.

Finalmente, es necesario considerar que algunos de las vulnerabilidades de software se deben a fallos inesperados en el entorno, provocados, por ejemplo, por un disco lleno. KLEE permite de manera opcional simular estos fallos del sistema de una forma controlada para ayudar a encontrar vulnerabilidades causadas por este tipo de fallos.

5.3. Herramientas

El software de KLEE se compone de una serie de herramientas por línea de comandos, entre las más importantes podemos destacar:

- **klee**: Es la herramienta de ejecución simbólica que interpreta el código LLVM IR, que se explica en la sección siguiente.
- **kleaver**: Se trata de el solucionador de restricciones que aplica simplificaciones matemáticas y optimizaciones sobre las restricciones. Kleaver se encarga de agruparlas en subconjuntos independientes y de gestionar estos subconjuntos y superconjuntos en caché, y si finalmente es necesario realiza las necesarias al solucionador SMT subyacente.
- **ktest-tool**: Se trata de una herramienta que permite leer el caso de test generado por KLEE en formato binario.
- **klee-stats**: Es un script en Python que permite presentar las estadísticas de la ejecución de KLEE.
- **klee-replay**: Permite lanzar una aplicación de forma nativa pasando los parámetros necesarios para reproducir un caso de test.
- **gen-random-bout**: Una herramienta que permite generar test aleatorios.

5.4. El intérprete KLEE

En esta sección se presenta el funcionamiento de KLEE como intérprete de LLVM IR [18], que puede ser llamado desde la línea de comandos. Se parte de un sencillo ejemplo, *"vul_exploit.c"*, de programa con una vulnerabilidad de memoria.

Listado 5.1: Programa vulnerable al *bufferoverflow*

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

int main(int argc, char *argv[]) {
    char *first, *second, *third;
```

```
    first = malloc(100);
    second = malloc(12);
    third = malloc(12);

    strcpy(first, argv[1]);
    free(first);
    free(second);
    free(third);
    return (0);
}
```

Normalmente, KLEE no necesita que los fuentes del programa sean modificados. Sin embargo, para conseguir el bitcode LLVM *"vul_exploit.c"* es necesario compilar con "*clang*", haciendo uso de la opción "-emit-llvm":

```
clang -emit-llvm -c vul_exploit.c -o vul_exploit.bc
```

Por defecto, KLEE trabaja sobre programas cerrados, esto es, autocontenidos, que no usan ningún código externo. Sin embargo, para que KLEE pueda trabajar como intérprete de LLVM IR sobre programas reales es necesario haberlo configurado, en el momento de su construcción, con soporte para un entorno de ejecución POSIX y la librería **uClibc**, modificada para su uso con KLEE. De esta forma estos componentes pueden ser habilitados en el momento de la ejecución.

De esta forma, KLEE necesita tener definiciones para todas las funciones externas que el programa pueda llamar. Por medio de la opción "*-libc=uclibc*" se indica la carga de la librería **uClibc** modificada y su enlazado con el código del programa en bitcode antes de lanzar la ejecución.

Adicionalmente, un programa necesita ejecutarse sobre un sistema operativo que le proporciona funciones de bajo nivel, como por ejemplo "*write()*". Es por esto que KLEE proporciona una implementación limitada de un entorno de ejecución POSIX adaptada para su funcionamiento con la librería **uClibc**. La opción "*-posix-runtime*" indica el enlazado previo, con esta implementación POSIX.

Ya solo queda lanzar KLEE sobre el bitcode generado, definiendo opcionalmente el número, tamaño y tipo de entradas simbólicas a utilizar para analizar el programa. En el ejemplo, se ha utilizado la opcin "*-sym-arg*" para indicar un argumento de entrada con una longitud máxima de 200:

```
$ klee --libc=uclibc -posix-runtime vul_exploit.bc -sym-arg 200
KLEE: NOTE: Using klee-uclibc : /home/klee/klee_build/klee/Release+Debug+
    Asserts/lib/klee-uclibc.bca
KLEE: NOTE: Using model: /home/klee/klee_build/klee/Release+Debug+Asserts/
    lib/libkleeRuntimePOSIX.bca
KLEE: output directory is "/home/klee/data/tfm/src/klee-out-0"
KLEE: Using STP solver backend
KLEE: WARNING ONCE: calling external: syscall(16, 0, 21505, 48281232) at /
    home/klee/klee_src/runtime/POSIX/fd.c:1044
KLEE: WARNING ONCE: calling __user_main with extra arguments.
KLEE: WARNING ONCE: Alignment of memory from call "malloc" is not modelled.
     Using alignment of 8.
KLEE: ERROR: /home/klee/klee_build/klee-uclibc/libc/string/strcpy.c:27:
    memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 9886
KLEE: done: completed paths = 101
KLEE: done: generated tests = 101
```

En las trazas resultantes puede observarse una serie de advertencias, que también quedarán almacenadas en el archivo "*warnings.txt*". Entre estas advertencias podemos encontrar:

- "*Calling external: syscall()*": Se está llamando a una función usada en el programa pero de la que no se tiene una definición. En este caso se llama a la función nativa "*syscall()*".
- "*Executable has module level assembly (ignoring)*": KLEE no puede interpretar las instrucciones en ensamblador, y en consecuencia las ignora.
- "*Undefined reference to function*": El programa contiene una llamada a una función no definida. Si el programa acaba haciendo una llamada a esta función, KLEE no podrá interpretarla y terminará la ejecución.
- "*Calling __user main with extra arguments*": La función "*__user_main()*" es realmente la función "*main()*" del programa que ha sido renombrada antes de enlazarla con **uClibc**. La advertencia, generalmente sin trascendencia, indica

que se ha llamado a la función con un número mayor de argumentos de los esperados por parte de **uClibc**, por ejemplo el puntero de entorno.

- "*Alignment of memory from call ... not modelled*": En este caso en la llamada "*malloc()*", indica que la modelización simbólica del tamaño de memoria no está soportada, con lo que se emplea un valor concreto, en este caso de 8.

Por otro lado, en las trazas generadas por KLEE se muestra la detección del desbordamiento de "*buffer*" que puede afectar a la instrucción "*strcpy()*", generando el caso de test que provoca el problema, en este caso en el archivo "*test000100.ptr.err*". Archivo que sigue la nomenclatura "*testN.TYPE.err*" para los casos de error, siendo N el número de test y TYPE el tipo de error detectado, entre los cuales están:

- **ptr**: Carga y almacenaje de posiciones de memoria inválidas.
- **free**: Doble liberación de memoria o liberación inválida, a través de la instrucción "*free()*".
- **abort**: Llamada del programa a la instrucción "*abort()*".
- **assert**: Fallo en una aserción.
- **div**: Detección de una división o módulo por cero.
- **user**: Existe un problema con el programa de entrada o el modo en que se está usando KLEE.
- **exec**: Existe un problema que impide que KLEE ejecute el programa (ej.: instrucción desconocida, llamada a un puntero de función inválido o llamada a una instrucción de ensamblador).
- **model**: KLEE no puede tener una precisión completa y solo está explorando una parte de los estados posibles del programa (caso por ejemplo de la función "*alloc()*" y la ausencia de soporte simbólico a su argumento de tamaño de memoria).

La ejecución de KLEE crea un nuevo directorio, "*klee-last*", donde se generan una serie de archivos de salida divididos en dos categorías: **archivos globales** y **archivos de rutas**.

Los **archivos globales** siempre se generan durante la ejecución de KLEE. Entre estos archivos podemos encontrar:

- **info**: Incluye información general sobre la ejecución de KLEE, como el comando exacto de la ejecución, el tiempo transcurrido, número de rutas exploradas, etc.

```
/home/jaoses/data/socle/klee/Release+Asserts/bin/klee --libc=uclibc -
    posix-runtime vul_exploit.bc -sym-arg 200
PID: 5291
Started: 2018-08-14 13:36:01
BEGIN searcher description
<InterleavedSearcher> containing 2 searchers:
RandomPathSearcher
WeightedRandomSearcher::CoveringNew
</InterleavedSearcher>
END searcher description
Finished: 2018-08-14 13:36:01
Elapsed: 00:00:00
KLEE: done: explored paths = 101
KLEE: done: avg. constructs per query = 4
KLEE: done: total queries = 102
KLEE: done: valid queries = 0
KLEE: done: invalid queries = 102
KLEE: done: query cex = 102

KLEE: done: total instructions = 10233
KLEE: done: completed paths = 101
KLEE: done: generated tests = 101
```

- **warnings.txt**: Contiene todas las advertencias emitidas por KLEE.
- **messages.txt**: Contiene el resto de mensajes emitidos por KLEE (ejemplo, los errores).
- **assembly.ll**: Contiene una versión legible del bitcode LLVM IR ejecutado por

KLEE, que incluso para un programa sencillo como el del ejemplo puede tener un tamaño considerable (1,2 MB).

```
; ModuleID = 'vul_exploit.bc'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
    i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0
    :64:64-f80:128:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"
...
; Function Attrs: nounwind uwtable
define i32 @__user_main(i32 %argc, i8** %argv) #0 {
  %argcPtr = alloca i32
  %argvPtr = alloca i8**
  store i32 %argc, i32* %argcPtr
  store i8** %argv, i8*** %argvPtr
  call void @klee_init_env(i32* %argcPtr, i8*** %argvPtr)
  %newArgc = load i32* %argcPtr
  %newArgv = load i8*** %argvPtr
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i8**, align 8
```

- **run.stats**: Un archivo de texto que contiene estadísticas de la ejecución, que pueden inspeccionarse con la herramienta "*klee-stats*".
- **run.istats**: Contiene estadísticas globales emitidas por KLEE para cada línea de código del programa.
- **all-queries.kquery**: Contiene todas las consultas generadas por KLEE para el solucionador SMT, en formato KQuery, antes de la etapa del solucionador SMT, por lo que no necesariamente son consultas que se pasen al solucionador. Es necesario utilizar la opción "*–use-query-log=all:kquery*" para generar este archivo.

```
...
# Query 2 -- Type: InitialValues, Instructions: 7799
array arg0[201] : w32 -> w8 = symbolic
array model_version[4] : w32 -> w8 = symbolic
```

```
(query [(Eq 1
             (ReadLSB w32 0 model_version))
         (Eq 0 (Read w8 0 arg0))]
        false []
        [arg0
         model_version])
#   OK -- Elapsed: 3.194809e-05
#   Solvable: true
#     arg0 = [0,0,0,0,0,  ...
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
#     model_version = [1,0,0,0]
...
```

KQuery es el lenguaje utilizado por KLEE para representar las expresiones de restricciones, justo antes de pasarlas al solucionador Kleaver. KQuery es capaz de representar fórmulas sin cuantificadores para vectores de bits y *arrays*, disponiendo de soporte directo para las operaciones sobre vectores de bits.

- **all-queries.smt2**: Contiene la misma información que "*all-queries.kquery*" en formato SMT-LIBv2. Para su generación es necesario utilizar la opción "*–use-query-log=all:smt2*".

```
...
; Query 2 -- Type: Validity, Instructions: 7801
(set-logic QF\_AUFBV )
(declare-fun arg0 () (Array (\_ BitVec 32) (\_ BitVec 8) ) )
(declare-fun model\_version () (Array (\_ BitVec 32) (\_ BitVec 8) ) )
(assert (and  (and  (=  (\_ bv0 8) (select  arg0 (\_ bv1 32) ) ) (=
    (\_ bv1 32) (concat  (select  model\_version (\_ bv3 32) ) (concat
     (select  model\_version (\_ bv2 32) ) (concat  (select  model\
    _version (\_ bv1 32) ) (select  model\_version (\_ bv0 32) ) ) ) )
    ) ) (=  false (=  (\_ bv0 8) (select  arg0 (\_ bv0 32) ) ) ) ) )
(check-sat)
(exit)
;   OK -- Elapsed: 1.467943e-03
;   Validity: 0
...
```

- **solver-queries.kquery**: Contiene todas las consultas que KLEE realiza al solucionador SMT después del proceso de optimización en formato KQuery. Se genera a través de la opción "*–use-query-log=solver:kquery*".
- **solver-queries.smt2**: Contiene la misma información que "*solver-queries.kquery*" en formato SMT-LIBv2.

Los **archivos de rutas** son generados por cada ruta del programa explorada. Estos son los siguientes:

1. **test<N>.ktest**: Contiene el test generado para la ruta correspondiente, que puede visualizarse con "*ktest-tool*". Si se quiere omitir la generación de estos archivos, puede hacerse a través de la opcin "*–no-output*". En el ejemplo, la inspección del archivo "*test000100.ktest*" con el caso del test en error muestra la información con una de las posibles cadenas pasadas como argumento que provocan el problema:

```
$ ktest-tool test000100.ktest
ktest file : 'test000100.ktest'
args       : ['vul_exploit.bc', '-sym-arg', '200']
num objects: 2
object    0: name: b'arg0'
object    0: size: 201
object    0: data: b'\x01\x01\x01\x01 ... \x00\x00\x00\x00'
object    1: name: b'model\_version'
object    1: size: 4
object    1: data: b'\x01\x00\x00\x00'
```

2. **test<N>.<error-type>.err**: Se crea para las rutas que resultan en error. Contiene información en texto plano sobre el tipo de error que se ha producido. Así, en el ejemplo, el archivo "*test000100.ptr.err*" contiene la información sobre el desbordamiento de buffer en la llamada a "*strcpy()*":

```
Error: memory error: out of bound pointer
File: /home/klee/klee\_build/klee-uclibc/libc/string/strcpy.c
Line: 27
assembly.ll line: 1002
```

```
Stack:
    #000001002 in strcpy (s1=40825392, s2=57115440) at /home/klee/klee
        \_build/klee-uclibc/libc/string/strcpy.c:27
    #100000191 in \_\_user_main (argc=3, argv=47457216) at /home/klee/
        data/tfm/src/vul_exploit.c:17
    #200001746 in \_\_uClibc_main (main=40134672, argc=3, argv
        =47457216, app\_init=0, app_fini=0, rtld\_fini=0, stack_end=0)
        at /home/klee/klee\_build/klee-uclibc/libc/misc/internals/\_\
        _uClibc_main.c:401
    #300003228 in main (=3, =47457216)
Info:
    address: 40825492
    next: object at 41317600 of size 12
        MO3132[12] allocated at \_\_user_main():  %6 = call noalias i8
            * @malloc(i64 12) #11, !dbg !2038
    prev: object at 40825392 of size 100
        MO3128[100] allocated at \_\_user_main():  %4 = call noalias
            i8* @malloc(i64 100) #11, !dbg !2036
```

3. **test<N>.kquery**: Contiene las restricciones asociadas con la ruta correspondiente, en formato KQuery. Su generación se activa por medio de la opción "*–write-kqueries*".

4. **test<N>.cvc**: Contiene la misma información que los archivos "*test¡N¿.kquery*", en formato CVC. Su generación se habilita a través de la opción "*–write-cvcs*".

5. **test<N>.smt2**: Contiene la misma información que los archivos "*test<N>.kquery*", en formato SMT-LIBv2. Se activa a través de la opción "*–write-smt2*".

5.5. API de desarrollo

La herramienta KLEE dispone de un API en C que permite incorporar la ejecuciíon simbóla en otros programas, definiendo las variables simbólicas o las restricciones sobre la condición de ruta. A través de este API, KLEE implementa un mecanismo de instrumentación de código, que se lanza en el momento de la ejecución del programa por parte del intérprete.

Las funciones del API de KLEE son llamadas "*intrínsecas*", ya que KLEE las trata internamente de manera especial. Entre ellas cabe destacar:

- *klee_make_symbolic()*: Permite marcar un espacio de memoria como simbólico. También se usa para inicializar los datos de test en el momento de usar la herramienta "*ktest-tool*".
- *klee_assume(<condición>)*: Define una condición que deben cumplir los estados que explora la ejecución simbólica, reduciendo el espacio de búsqueda.
- *klee_prefer_cex(<objeto>, <condición>)*: Esta función permite indicar a KLEE la preferencia por ciertos valores en el moment de generar los casos de test para el objeto. La condición define esa preferencia y se aplica siempre que sea posible.
- *klee_assert*: Se trata de una macro que en caso de error invoca a una función interna para su manejo.

Para poder usar estas funciones, el programa a analizar deben ser enlazadas estáticamente con la librería "**libkleeRunTest**", de la que forman parte.

5.6. Limitaciones

A pesar de la versatilidad de KLEE, esta herramienta también presenta algunas limitaciones para su la aplicaciíon práctica, como de hecho ha sido indicado en foros especializados [19]:

1. El modelo de memoria utilizado por el intérprete de LLVM IR consume muchos recursos de memoria y no está bien optimizado en tiempo. Para cada ruta de ejecución KLEE mantiene un espacio de direcciones que contiene un *stack* para variables locales, un *heap* para las variables globales y las dinámicas. Cada variable (local o global) se encapsula en un objeto *MemoryObject*, que mantiene metadados relativos a la variable, como puede ser: dirección, tamaño e información sobre la reserva de memoria. Esto hace que el tamaño de memoria de cada variable se incremente con el tamaño del objeto *MemoryObject* correspondiente. Cada vez que KLEE accede a una variable verifica si está en alguno

de los objetos *MemoryObject* creados. Si bien se trata de un método fácil para implementar la ejecuciíon simbólica, no es óptimo.

2. No dispone apenas modelos para los componentes del sistema: El único que viene modelado es el sistema de archivos. Otros, como pueden ser los *sockets* y el multihilo no están soportados. Cuando KLEE hace llamadas al sistema ("*systemcall()*") sobre los componentes no modelados, o bien termina la exploración de la ruta y lanza una alerta, o redirige la llamada al sistema operativo. Esto último tiene varios problemas: los valores simbólicos necesitan concretizarse y con esto se pierden rutas, además, podría existir solapamiento de rutas.

3. KLEE no trabaja directamente con binarios, sino que necesita LLVM IR. Esto obliga a disponer de los fuentes y crear una compilaciíon a medida del programa a probar.

Si bien, la comprobación de una de ellas requiere un conocimiento profundo de la herramienta, las otras dos han sido en buena medida verificadas por este estudio.

Capítulo 6

Extendiendo KLEE

En este capítulo se abordan una serie de aspectos cuya consideración representa una mejora de la herramienta KLEE en su aplicación práctica. Se exponen las modificaciones sobre el código fuente de KLEE que se han realizado y los resultados conseguidos.

6.1. Instalación

Actualmente existen básicamente dos posibilidades para poder instalar KLEE: utilizar una imagen de Docker, que se puede descargar de su web [18], y que actualmente es un "*Ubuntu 14.04.5 LTS*", o compilar directamente el código fuente sobre el ordenador deseado. Se han probado ambas opciones, y la opciíon de compilación sobre un sistema operativo *Ubuntu 16.04.5 LTS*, aunque más compleja, ha resultado más flexible, puesto que permite una mejor actualizaciíon de las herramientas sobre las que se apoya KLEE.

Sin embargo, para el procedimiento la compilación de KLEE, se ha demostrado más adecuada la información proporcionada en el URL [20], proporcionada por la Universidad Técnica de Munich, que la disponible en la web oficial de KLEE. Lo que muestra una deficiencia en la gestión del código fuente de la herramienta.

Por tanto, los pasos que se han dado para poder instalar KLEE sobre Ubuntu 16.04 han sido los siguientes:

- Instalar las herramientas necesarias, con el comando "*apt-get install*", si es

necesario:

```
$ sudo apt-get install bc bison build-essential cmake curl flex git
    libboost-all-dev libcap-dev libncurses5-dev python-minimal python-
    pip subversion unzip zlib1g-dev
```

- Construir la infraestructura LLVM a partir de su repositorio de código fuente de *subversion*, en versión 3.4.2. Para este paso es necesario también Python 2, para lanzar el comando "*make check-all*" por lo que se indicará su ruta en la configuración (opción "–with-python").

```
$ svn co https://llvm.org/svn/llvm-project/llvm/tags/RELEASE_342/final
     llvm
$ svn co https://llvm.org/svn/llvm-project/cfe/tags/RELEASE_342/final
    llvm/tools/clang
$ svn co https://llvm.org/svn/llvm-project/compiler-rt/tags/
    RELEASE_342/final llvm/projects/compiler-rt
$ svn co https://llvm.org/svn/llvm-project/libcxx/tags/RELEASE_342/
    final llvm/projects/libcxx
$ svn co https://llvm.org/svn/llvm-project/test-suite/tags/RELEASE_342
    /final/ llvm/projects/test-suite

$ rm -rf llvm/.svn
$ rm -rf llvm/tools/clang/.svn
$ rm -rf llvm/projects/compiler-rt/.svn
$ rm -rf llvm/projects/libcxx/.svn
$ rm -rf llvm/projects/test-suite/.svn

$ cd llvm
$ ./configure --enable-optimized --disable-assertions --enable-targets
    =host --with-python="/usr/bin/python2"
$ make -j `nproc`
$ make -j `nproc` check-all
$ cd ..
```

- Construir MiniSAT a partir del código fuente disponible en su repositorio "*git*":

```
$ git clone --depth 1 https://github.com/stp/minisat.git
$ rm -rf minisat/.git
$ make
$ cd ..
```

- Instalar STP versión 2.2.0 a partir de su código fuente de "*git*".

```
$ git clone --depth 1 --branch stp-2.2.0 https://github.com/stp/stp.
    git
$ rm -rf stp/.git
$ cd stp
$ mkdir build
$ cd build
$ cmake \
   -DBUILD_STATIC_BIN=ON \
   -DBUILD_SHARED_LIBS:BOOL=OFF \
   -DENABLE_PYTHON_INTERFACE:BOOL=OFF \
   -DMINISAT_INCLUDE_DIR="../../minisat/" \
   -DMINISAT_LIBRARY="../../minisat/build/release/lib/libminisat.a" \
   -DCMAKE_BUILD_TYPE="Release" \
   -DTUNE_NATIVE:BOOL=ON ..
$ make -j `nproc`
$ cd ../..
```

- Instalar a partir del código fuente "*klee-uclib*" en "*git*": uClibc y modelo de entorno POSIX.

```
$ git clone --depth 1 --branch klee_uclibc_v1.0.0 https://github.com/
    klee/klee-uclibc.git
$ rm -rf klee-uclibc/.git
$ cd klee-uclibc
$ ./configure \
 --make-llvm-lib \
 --with-llvm-config="../llvm/Release/bin/llvm-config" \
 --with-cc="../llvm/Release/bin/clang"
$ make -j `nproc`
$ cd ..
```

- Instalar Z3 en versión 4.5.0 a partir de su código fuente del repositorio "*git*".

```
$ git clone --depth 1 --branch z3-4.5.0 https://github.com/Z3Prover/z3
    .git
$ rm -rf z3/.git

$ cd z3
$ python scripts/mk_make.py
$ cd build
$ make -j `nproc`
```

- Adecuar el emplazamiento de algunos archivos "*header*" y librerías dinámicas, para facilitar la compilición de KLEE.

```
$ mkdir -p ./include
$ mkdir -p ./lib

$ cp ../src/api/z3.h ./include/z3.h
$ cp ../src/api/z3_v1.h ./include/z3_v1.h
$ cp ../src/api/z3_macros.h ./include/z3_macros.h
$ cp ../src/api/z3_api.h ./include/z3_api.h
$ cp ../src/api/z3_ast_containers.h ./include/z3_ast_containers.h
$ cp ../src/api/z3_algebraic.h ./include/z3_algebraic.h
$ cp ../src/api/z3_polynomial.h ./include/z3_polynomial.h
$ cp ../src/api/z3_rcf.h ./include/z3_rcf.h
$ cp ../src/api/z3_fixedpoint.h ./include/z3_fixedpoint.h
$ cp ../src/api/z3_optimization.h ./include/z3_optimization.h
$ cp ../src/api/z3_interp.h ./include/z3_interp.h
$ cp ../src/api/z3_fpa.h ./include/z3_fpa.h
$ cp ../src/api/c++/z3++.h ./include/z3++.h
$ cp libz3.so ./lib/libz3.so
$ cd ../..
```

- Instalar KLEE a partir de su código fuente en versión 1.4.0, disponible en su repositorio "*git*". Este paso necesita de rutas absolutas en la configuración.

```
$ git clone --depth 1 --branch v1.4.0 https://github.com/klee/klee.git
$ rm -rf klee/.git

$ export BUILDDIR=`pwd`
$ cd klee
$ ./configure \
   LDFLAGS="-L$BUILDDIR/minisat/build/release/lib/" \
   --with-llvm=$BUILDDIR/llvm/ \
   --with-llvmcc=$BUILDDIR/llvm/Release/bin/clang \
   --with-llvmcxx=$BUILDDIR/llvm/Release/bin/clang++ \
   --with-stp=$BUILDDIR/stp/build/ \
   --with-uclibc=$BUILDDIR/klee-uclibc \
   --with-z3=$BUILDDIR/z3/build/ \
   --enable-cxx11 \
   --enable-posix-runtime

$ make -j `nproc` ENABLE_OPTIMIZED=1
```

- Desplazar la librería "*libz3.so*" donde KLEE pueda encontrarla y lanzar los test.

```
$ cp ../z3/build/lib/libz3.so ./Release+Asserts/lib/

$ make -j `nproc` check
$ cd ..
```

- Finalmente es recomendable crear alias para acceder a los binarios más importantes en "*∼/build/klee/Release+Asserts/bin*" y "*∼/build/llvm/Release/bin*".

Como puede observarse la instalación de KLEE es farragosa, no está muy perfeccionada, y algunas de las librerías como MiniSAT son antiguas.

6.2. Interacción con el entorno

La capacidad de interacción de una herramienta de análisis con el entorno (sistema operativo, *sockets*, sistemas de entrada/salida, ...) es una de las características más importantes para la aplicación práctica.

En el caso de KLEE, la interacción con el entorno se realiza por medio del empleo de la librería **klee-ulibc** y el modelado simbólico de un entorno POSIX. Ya que la modelización que se hace de un entorno POSIX es limitada, existen funciones en **klee-uclibc** que realizan llamadas a componentes no modelizados, es el caso de la función "*syslog()*". Esto puede bloquear el análisis del código cuando este hace uso de dichas funciones. Para desbloquear este análisis puede emplearse el uso de funciones "***mock***". Se pierde el análisis de esta función pero se continúa con el análisis del resto de código.

En el ejemplo siguiente se tiene un sencillo ejemplo de la función "*syslog()*":

Listado 6.1: Ejemplo de llamada a *syslog()*

```
#include <string.h>
#include <syslog.h>

#define MAX_SIZE 50

void write_log(char *message) {
```

```
    openlog("slog", LOG_PID|LOG_CONS, LOG_USER);
    syslog(LOG_NOTICE, message);
    closelog();
}

int main(int argc, char *argv[]) {
  char buffer[MAX_SIZE];

  if (argc > 0) {
    strcpy(buffer, argv[1]);
    write_log(buffer);
  }
}
```

Si utilizamos KLEE sobre el anterior programa, la ejecución se interrumpe debido a que no está soportado la llamada a una instrucción de ensamblador.

```
$ symtool simple_syslog.bc --sym-arg 3
UKLEE_HOME: /home/klee/socle/klee/Release+Asserts/bin
KLEE: NOTE: Using klee-uclibc : /home/klee/socle/klee/Release+Asserts/lib/
    klee-uclibc.bca
KLEE: NOTE: Using model: /home/klee/socle/klee/Release+Asserts/lib/
    libkleeRuntimePOSIX.bca
KLEE: output directory is "/home/klee/tfm/obj/klee-out-0"
KLEE: Using STP solver backend
KLEE: WARNING ONCE: function "__libc_connect" has inline asm
KLEE: WARNING ONCE: function "socket" has inline asm
KLEE: WARNING ONCE: calling external: syscall(16, 0, 21505, 57710496) at /
    home/klee/socle/klee/runtime/POSIX/fd.c:1044
KLEE: WARNING ONCE: calling __user_main with extra arguments.
KLEE: WARNING ONCE: Alignment of memory from call "malloc" is not modelled.
     Using alignment of 8.
KLEE: WARNING ONCE: __syscall_rt_sigaction: silently ignoring
KLEE: ERROR: /home/klee/socle/klee-uclibc/libc/inet/socketcalls.c:362:
    inline assembly is unsupported
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 50572
KLEE: done: completed paths = 4
KLEE: done: generated tests = 1
```

Con una versión *mock* de la función "*syslog()*" se puede evitar este problema. Es-

to puede hacerse a través de una variable de entorno, de forma que si está activada no se realice ninguna acción. Para conseguir esto, se ha modificado el archivo "*syslog.c*" presente en el directorio **klee-uclibc**, y se han aãdido los archivos "*mock.h*" y "*mock.c*" con el fin de implementar las funciones de ayuda.

Si lanzamos KLEE con la variable indicadora del "*mock*", el problema anterior desaparece:

```
$ symtool simple_syslog.bc --sym-arg 3
UKLEE_HOME: /home/klee/socle/klee/Release+Asserts/bin
KLEE: NOTE: Using klee-uclibc : /home/klee/socle/klee/Release+Asserts/lib/
    klee-uclibc.bca
KLEE: NOTE: Using model: /home/klee/socle/klee/Release+Asserts/lib/
    libkleeRuntimePOSIX.bca
KLEE: output directory is "/home/klee/tfm/obj/klee-out-1"
KLEE: Using STP solver backend
KLEE: WARNING ONCE: function "__libc_connect" has inline asm
KLEE: WARNING ONCE: function "socket" has inline asm
KLEE: WARNING ONCE: calling external: syscall(16, 0, 21505, 63584304) at /
    home/klee/socle/klee/runtime/POSIX/fd.c:1044
KLEE: WARNING ONCE: calling __user_main with extra arguments.
KLEE: WARNING ONCE: Alignment of memory from call "malloc" is not modelled.
     Using alignment of 8.
KLEE: WARNING ONCE: calling external: printf(68423488, 81939928, 66696464)
    at /home/klee/socle/klee-uclibc/libc/misc/mock/mock.c:23
Mock at syslog is true
Mock at syslog is true
Mock at syslog is true
Mock at syslog is true

KLEE: done: total instructions = 19911
KLEE: done: completed paths = 4
KLEE: done: generated tests = 4
```

6.3. Un *framework* de test simbólico

Una de las etapas del ciclo de desarrollo de software es la fase de test, en sus distintas variantes: test unitarios, test de integraciíon, test de no regresión, etc. Una mejora

de la calidad en el ciclo de desarrollo de software es la posibilidad de añadir en la etapa de test de seguridad una fase de búsqueda de vulnerabilidades, y es en este campo donde una herramienta como KLEE puede ser adecuada.

Sin embargo, KLEE está concebido para lanzar un análisis sobre un programa, no una batería de ellos. Por esto, para mejorar la habilitación de KLEE con este objetivo es necesario:

1. Modificar el motor de ejecución simbólica KLEE para poder lanzar una función individual definida, pasada como parámetro.

2. Crear un paquete de utilidades que permitan ayudar en la definición de casos de test o interrogar a la herramienta.

6.3.1. Modificaciones sobre KLEE

KLEE es una herramienta que inicia su ejecución con la carga de la función "*main()*" del programa. Sin embargo, en el diseño de un caso de test, resulta interesante probar solo una parte del programa, donde una determinada función es el punto de entrada. Por tanto es necesario que KLEE sea capaz de iniciar la ejecución a partir de otra función que no sea "*main()*".

De forma simplicada el código fuente de KLEE posee el núcleo de su intérprete bajo el directorio "*lib*" y el directorio "*tools*" recoge el punto de arranque de sus distintas herramientas (klee, kleaver, ktest-tool, ...). Por esto, para conseguir lanzar una función que es pasada como parámetro es necesario modificar el fichero "**main.cpp**" del directorio "*tools/klee*". Se ha optado por crear un nuevo directorio "tools/klee-ut", con lo que existe una nueva aplicación "**klee-ut**" ("*KLEE Unit Testing*") cuya compilación se integra fácilmente en la compilación global de KLEE. El listado 6.2 muestra el código C++ del reemplazo del nombre de función que se hace para "**klee-ut**" en el archivo "*main.cpp*":

Listado 6.2: Reemplazo de la función "*main()*"

```
static llvm::Module *changeMainFunction(llvm::Module *mainModule,
                                        StringRef functionName) {
  Function *userMainFn = mainModule->getFunction(EntryPoint);
  assert(userMainFn && "unable to get user main");
```

```
  cl::opt<std::string>
  SwapEntryPoint("swap-entry-point",
   cl::desc("Consider the function as the entrypoint"),
   cl::init(functionName.data()));

  Function *userTestFn = mainModule->getFunction(SwapEntryPoint);

  if ((userMainFn != NULL) && (userTestFn != NULL)) {
    klee_message("Replacing %s function with main",
                                  functionName.data());
    userMainFn->setName("__true_user_main");
    userTestFn->setName("main");
  }

  return mainModule;
}
```

Como puede apreciarse, KLEE accede a la manipulación de las instrucciones LLVM IR apoyándose sobre el API que ofrece LLVM. Así en el código se define una nueva variable "*SwapEntryPoint*" a partir del nombre de la función de reemplazo, que permite retomar el objeto *llvm* de tipo "*Functión*" correspondiente, y cambiar su nombre por el de "*main*". Al mismo tiempo se consigue una referencia a la verdadera función "*main()*", reemplazando su nombre por "*__true_user_main*", quedando de esta forma anulada.

Hay que tener en cuenta que KLEE realiza el reemplazo de la función "*main()*" en el momento de enlazar con la librería **uclibc**. En este cambio, la función "*main()*" queda reemplazada por "*__user_main*", siendo la función de entrada real "*__uClibc_main*". Por tanto, el cambio que indica el código del listado 6.2 se hace antes de que ocurra este proceso.

De esta forma, se ha conseguido incorporar a KLEE dos opciones más en la línea de comandos, con el fin de:

- Lanzar la ejecución de una función determinada (opción "*–run-function*"), especificando a continuación el nombre de la función.
- Lanzar la ejecución de un conjunto de funciones (opción "*–run-test-suite*"). Este caso es una generalización del anterior, donde se pasa como parámetro un archivo de texto con los nombres de las funciones a lanzar.

Para incorporar la posibilidad de ejecutar un conjunto de funciones y hacer que los cambios sean lo menos impactantes posible para el resto de código de la herramienta, se ha creado una nueva función "*main()*", renombrando la original como "*main_orig()*", que hace mucho más fácil la gestión de las nuevas opciones (ver listado 6.3).

Listado 6.3: Nueva función "*main()*"

```
int main(int argc, char **argv, char **envp) {
  int i, res, taille = 0;
  pid_t child_pid, wpid;
  int status = 0;
  char buffer[BUFFER_SIZE];

  pTList pList = detectTestFunction(argc, argv);
  if (pList == NULL) {
    return main_orig(argc, argv, envp, NULL);
  }

  taille = pList->size;
  for (i=0; i<taille; i++) {
    pop_element(pList, buffer, &res);
    printf("Test: %s\n", buffer);
    if (res < 0) {
      continue;
    }
    if ((child_pid = fork()) == 0) {
      main_orig(argc, argv, envp, buffer);
      exit(0);
    }
    while ((wpid = wait(&status)) > 0);
  }
  return 0;
}
```

Como puede observarse en el listado 6.3, para iniciar el proceso de ejecución simbólica se ha utilizado la instrucción "*fork()*", esto es, la creación de un nuevo proceso hijo por cada ejecución tratada. No es casual, el simple hecho de iterar directamente sobre cada ejecución produce la caída de la aplicación por corrupción de la memoria, lo que demuestra que KLEE no gestiona esta de manera adecuada. Con la creación de un nuevo proceso hijo se consigue sortear este problema, ya que utiliza un nuevo espacio de memoria.

Por otra parte, la lectura del archivo con las funciones a lanzar se realiza en la función "*detectTestFunction()*" que inicia una lista con las entradas del archivo, o con el nombre de una sola de la función en el caso de usar la opción para lanzar una función específica (ver listado 6.4).

Listado 6.4: Función detectTestFunction()

```
pTList detectTestFunction(int argc, char **argv) {
  int k;
  pTList pList = NULL;

  for (k = 0; k < argc; k++) {
    if ((strcmp(argv[k], "-run-function") == 0)
          || (strcmp(argv[k], "--run-function") == 0)) {
      if (++k == argc) {
         klee_message("--run-function expects argument <function-name>");
         break;
      }
      pList = init_stack();
      push_element(pList, argv[k++]);
      break;
    } else if ((strcmp(argv[k], "-run-test-suite") == 0)
                || (strcmp(argv[k], "--run-test-suite") == 0)) {
      if (++k == argc) {
         klee_message("--run-test-suite expects <test-suite-file>");
         break;
      }

      pList = get_test_suite(argv[k++]);
      dump_stack_suite(pList);
    }
  }
  return pList;
}
```

La implementación de la lista dinámica se ha hecho en otro archivo fuente ("*KleeUt.c*") incorporado al directorio "*lib/Core*".

Finalmente, se ha aadido un nuevo "*header*" ("*klee-ut.h*") en el directorio "*include*" con la definición de las funciones y estructuras necesarias.

6.3.2. Utilidades de ayuda

Para tener un mínimo "*framework*" de test es necesario poder estructurar los casos de test en el programa a examinar. KLEE provee un API de programación que puede ser usado en el código fuente del programa. Esto es, modificando el código fuente del programa original. Sin embargo, es deseable que el código del programa original no sea modificado y que se definan casos de test en archivos aparte.

Por tanto, es necesario utilizar el enlazado de código objeto de los archivos que contienen el código del programa y los que contienen el código de los test. Además, tenemos que considerar que se está trabajando con bitcode en LLVM IR, y no con código binario nativo, por lo que el manejo de "*clang*" de forma estándar no es suficiente.

Para conseguir el objetivo deseado es necesario instalar y utilizar el paquete de utilidades de WLLVM ("*Whole Program LLVM*")[21]. Estas utilidades, básicamente envoltorios en Python sobre las herramientas del proyecto LLVM, permiten generar código unificado en un solo programa sin modificar sus fuentes en C o C++. Lo realizan siguiendo los pasos siguientes, que habrá que considerar en el momento de crear los archivos *Makefile* para los test:

1. Compilan normalmente los archivos, generando el *bitcode* LLVM y guardándo la localización de este *bitcode* en una sección especial dedicada del código objeto. Se utiliza el compilador "*clang*", lo que necesita de la definición y exportación de la variable de entorno LLVM_COMPILER con dicho valor.

2. En el momento del enlazado, también con "*clang*", se concatenan las distintas secciones del *bitcode*.

3. En último lugar se extrae el contenido de la sección dedicada final y se enlaza todo el *bitcode* en un solo archivo. Para ello se necesita utilizar la herramienta de LLVM "*extract-bc*" sobre el binario generado.

Por otro lado, se ha escrito una pequea librería en C y Python 3 ("*Symbolic Tools*"), que permite hacer más compacto el código C de test a través de macros y que, por otro lado, proporciona una envoltura de la línea de comandos para "**klee-ut**"

a través de "*scripts*" Python. De esta forma, se dispone de los siguientes "*scripts*" Python:

- *symtool.py*: Un envoltorio sobre "**klee-ut**" para acortar y facilitar el uso de la línea de comandos.
- *simple_alnalysis.py*: Implementa un simple análisis estático del código *bitcode*, proporcionando el número de argumentos del programa y el máximo tamaño de *buffer* reservado que se ha utilizado. Para obtener esto realiza una búsqueda con expresiones regulares sobre el "*bitcode*" desensamblado previamente a un archivo de texto legible.
- *make_suite.py*: Crea el archivo con las funciones de test que hayan sido definidas en el código fuente de los test. Este archivo se pasa como argumento a "**klee-ut**". Utiliza expresiones regulares para conseguir esto.
- *setenv.sh*: Se trata de un "*script*" Bash que inicializa las variables de entorno necesarias y define una serie de alias que facilitan el manejo de los "*scripts*".

Para instalar esta pequeña librería ("*Symbolic Tools*") basta con definir en el archivo "*.bashrc*" su ubicación y lanzar el script "*setenv.sh*":

```
$ cat .bashrc
# SYM_TOOLS HOME
export SYM_TOOLS_HOME=/home/klee/symbolic_tools
. $SYM_TOOLS_HOME/setenv.sh
```

De esta forma, en el listado 6.5 puede verse el aspecto que tiene un test para KLEE (herramienta "*klee-ut*") en un archivo fuente independiente.

Listado 6.5: Test de ejecución simbólica, "*test_abs.c*"

```
#include <stdio.h>
#include <klee/klee.h>
#include "abs.h"
#include "sym_tools.h"
#include "sym_tools.inc"
```

```
#define SIZE 4

_Begin_Test(absValue)
  char arr[SIZE];
  int num;

  klee_make_symbolic(arr, sizeof arr, "in-arr");

  ASSUME_NUMERIC(SIZE, arr)

  num = absVal(arr);
  printf("Test abs value: %d\n", num);
_End_Test
```

En el código se utiliza la macro definida "_Begin_Test()" que expande el nombre de la función de forma que tiene el mismo tipo y número de parámetros que una función "*main()*". Esto es necesario para poder reemplazar el nombre de la función en KLEE sin que haya problemas.

Para conseguir el archivo con las funciones de test basta lanzar el comando "*make-suite*" de la librería de utilidades creada:

```
$ make-suite test_abs.c
Looking for test functions in test_abs.c
Generating test_abs.ts ...

$ cat test_abs.ts
test_absValue
test_absValue_small
```

6.4. Resultados

6.4.1. Creando una batería de test

Un ejemplo sencillo de la aplicación de test independientes puede hacerse con el código del listado 6.6, un ejemplo de cálculo de valor absoluto.

Listado 6.6: Valor absoluto "*abs.c*"

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

int absVal(char *str) {
  int x;
  if (str == NULL) {
    printf("Null parameter to absVal()\n");
    return 0;
  }

  x = atoi(str);
  if (x < 0 )
    return (-1) * x;

  if (x > 0)
    return x;

  return 0;
}

int main(int argc, char* argv[]) {
    int num;
    char message[50];

    if (argc != 2) {
      printf("Number of input arguments should be 1\n");
      exit(0);
    }

    num = absVal(argv[1]);
    sprintf(message, "Abs value: %d\n", num);
    return 0;
}
```

En el caso presentado, es importante el espacio de búsqueda para el éxito de la ejecución simbólica, como se indica en [6]. Esto se hace en el test del listado 6.5 con la macro "ASUME_NUMERIC()" que fuerza la búsqueda de cadenas numéricas terminada en NULL.

Si lanzamos "*symtool*" sobre el *bitcode* completo "*abs.bc*", la opción "*–run-test-suite*" y el archivo con las funciones extraídas "*test_abs.ts*", las trazas resultantes son las siguientes:

```
$ symtool abs.bc --run-test-suite test_abs.ts
UKLEE_HOME: /home/klee/socle/klee/Release+Asserts/bin
Test: test_absValue
KLEE: Replacing test_absValue function with main
KLEE: NOTE: Using klee-uclibc : /home/klee/socle/klee/Release+Asserts/lib/
    klee-uclibc.bca
KLEE: NOTE: Using model: /home/klee/socle/klee/Release+Asserts/lib/
    libkleeRuntimePOSIX.bca
KLEE: output directory is "/home/klee/tfm-test/bin/klee-out-0"
KLEE: Using STP solver backend
KLEE: WARNING ONCE: calling external: syscall(16, 0, 21505, 39815904) at /
    home/klee/socle/klee/runtime/POSIX/fd.c:1044
KLEE: WARNING ONCE: calling __user_main with extra arguments.
Test abs value: 373
Test abs value: 36
...
Test abs value: 989
Test abs value: 996
Test abs value: 995
Test abs value: 997
Test abs value: 998
Test abs value: 999
...
KLEE: done: total instructions = 395747
KLEE: done: completed paths = 1331
KLEE: done: generated tests = 1331
Test: test_absValue_small
KLEE: Replacing test_absValue_small function with main
KLEE: NOTE: Using klee-uclibc : /home/klee/socle/klee/Release+Asserts/lib/
    klee-uclibc.bca
KLEE: NOTE: Using model: /home/klee/socle/klee/Release+Asserts/lib/
    libkleeRuntimePOSIX.bca
KLEE: output directory is "/home/klee/tfm-test/bin/klee-out-1"
KLEE: Using STP solver backend
KLEE: WARNING ONCE: calling external: syscall(16, 0, 21505, 39815856) at /
    home/klee/socle/klee/runtime/POSIX/fd.c:1044
KLEE: WARNING ONCE: calling __user_main with extra arguments.
Test 2 abs value: 2
...
Test 2 abs value: 8
Test 2 abs value: 9
...
```

```
KLEE: done: total instructions = 7079
KLEE: done: completed paths = 11
KLEE: done: generated tests = 11
```

Como puede observarse, KLEE es lanzado dos veces, una por cada función de test, comprobándose que efectivamente la ejecución se deriva en las funciones de test, y finalmente obtenemos dos directorios con los resultados: "*klee-out-0*" y "*klee-out-1*".

Por otro lado, en el caso de los test lanzados, la ejecución simbólica discurre dentro del espacio de seguridad del "buffer" empleado por el programa, gracias a las restricciones sobre este espacio que se han definido. Sin embargo, si se quiere la herramienta sea capaz de encontrar la vulnerabilidad en este caso, debe ser parametrizada para poder explorar un espacio más amplio.

Es aquí donde un sencillo análisis estático puede ayudar a definir el espacio de búsqueda adecuado, dando el número de parámetros y el máximo valor de tamaño de "*buffer*" de memoria reservado. En este caso se puede utilizar la herramienta ' simple_analysis.py" a través de su alias, para obtener los datos:

```
$ simple-analysis .abs.o.bc
Maximum number of arguments: 1
Maximum size of allocation: 8
Maximum buffer size suggested: 9
```

Con estos datos, se puede definir una función de test que sobrepase el máximo valor de reserva de memoria. En [6] se da un valor heurístico de un 10 % por encima del máximo valor de reserva estática de memoria. Se trata de hecho de acotar el espacio de búsqueda al estrictamente necesario para probar el comportamiento del programa. Un valor mayor incrementa la complejidad de la búsqueda sin tener influencia sobre los resultados.

Listado 6.7: Caso de test para "buffer overflow"

```
_Begin_Test(absValue_over)
  char arr[9];
  int num;

  klee_make_symbolic(arr, sizeof arr, "in3-arr");
```

```
  ASSUME_NUMERIC(9, arr)

  num = absVal(arr);
  printf("Test 3 abs value: %d\n", num);
_End_Test
```

Al lanzar la utilizad simbólica sobre esta función, se detecta el desbordamiento de "*buffer*", como indican las trazas siguientes:

```
$ symtool abs.bc --run-function test_absValue_over
Command: /home/klee/socle/klee/Release+Asserts/bin/klee-ut --libc=uclibc --
    posix-runtime abs.bc --run-function test_absValue_over
Test: test_absValue_over
KLEE: Replacing test_absValue_over function with main
...
KLEE: WARNING ONCE: calling __user_main with extra arguments.
KLEE: ERROR: /home/klee/socle/klee-uclibc/libc/string/strcpy.c:27: memory
    error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
```

6.4.2. Test de Coreutils

Quizá uno de los paquetes de software más utilizado como modelo de estudio de las distintas metodologás dentro de la ejecución simbólica sea Coreutils ("*GNU Core Utilities*"). Para este trabajo también se han hecho algunas pruebas para comprobar el funcionamiento de KLEE.

Construcción de Coreutils

Se ha construido Coreutils a partir del código fuente en versión 8.30. Para esto, es necesario también, haber instalado previamente el paquete WLLVM para poder utilizar KLEE en su análisis. A partir de los fuentes, los pasos para obtener el código *bitcode* de Coreutils son los siguientes:

1. Definir la variable de entorno LLVM_COMPILER, con el valor "*clang*":

   ```
   $ export LLVM_COMPILER=clang
   ```

2. Desempaquetar los fuentes, crear el subdirectorio "*obj-llvm*" y configurar, se ha utilizado la opción "–disable-nls" para deshabilitar el soporte de internacionalización NLS ("*Native Language Support*") que añade información no necesaria para los test.

```
$ mkdir obj-llvm
$ cd obj-llvm
obj-llvm$ CC=wllvm ../configure --disable-nls CFLAGS="-g"
```

3. A continuación se lanza el comando "*make*" de la forma siguiente:

```
$ CC=wllvm make
```

4. Finalmente, se extrae el "*bitcode*" (*.bc) de los binarios generados por medio del comando:

```
$ find . -executable -type f | xargs -I '{}' extract-bc '{}'
```

Espacio de búsqueda

Como ha podido comprobarse el espacio de búsqueda en la ejecución simbólica es muy importante. Se puede utilizar Coreutils para ilustrar este hecho, por ejemplo con el programa "*echo*". Así, lanzando la ejecución simbólica para distintas longitudes se puede apreciar la influencia del espacio de búsqueda. El comando utilizado ha sido:

```
$ symtool echo.bc --sym-arg <longitud>
```

Con ayuda de la herramienta de estadísticas de KLEE ("*klee-stats*") se ha obtenido los datos del cuadro 6.1 con la cobertura de instrucciones LLVM exploradas (valor ICov(%)), la cobertura de ramas (valor BCov(%)) y el porcentaje de uso del solucionador SMT (valor TSolver(%), frente a la longitud de la cadena de entrada (valor L).

En este cuadro 6.1, se incluyen los datos para el caso de utilizar la opción de optimización, "–*optimize*" y el caso de no hacerlo. El empleo de la opción ("–*optimize*") de KLEE lanza la optimización dentro de LLVM para eliminar el código muerto proveniente de las librerías utilizadas en la construcción del ejecutable.

Cuadro 6.1: Cobertura de instrucciones y tiempo del solucionador SMT

L	$ICov$ (%)	$BCov$ (%)	$TSolver$ (%)	$ICov$ (%) opt.	$BCov$ (%) opt.	$TSolver$ (%) opt.
0	10,42	6,51	9,39	15,23	11,07	18,13
1	10,95	6,93	27,25	16,37	12,28	50,73
2	11,21	7,39	51,47	17,04	13,41	70,69
3	11,21	7,39	50,81	17,04	13,49	70,08
4	11,21	7,39	38,43	17,04	13,49	65,07
5	11,21	7,39	30,43	17,04	13,49	55,16
6	11,97	8,02	17,99	18,72	14,94	40,28
7	11,97	8,02	10,21	18,72	14,94	27,41
8	11,97	8,02	6,33	18,72	14,94	18,43
9	12,85	8,31	4,54	19,8	15,59	14,22
10	12,85	8,35	3,72	19,8	15,67	11,12

Como puede apreciarse en los datos y el diagrama 6.1, la optimización logra mejorar la cobertura de instrucciones alrededor de un 6-7 %, sin embargo, no parece lograr completamente la eliminación todo el código muerto, y este sigue teniendo una influencia apreciable.

Aún así, se ha constatado un aumento considerable de la velocidad de ejecución con el uso de estas optimizaciones. Esta mejora se produce en la exploración de las rutas y es una prueba de la eliminación de código muerto, realizada por parte de LLVM. De esta forma, como muestra el diagrama 6.2, el porcentaje de tiempo que se consume en el solucionador SMT es más alto en el caso de utilizar la optimización.

También, en el diagrama 6.2 se aprecia que conforme aumenta el espacio de búsqueda el porcentaje de tiempo del solucionador disminuye, ya que el optimizador de KLEE puede evitar las consultas repetitivas, con lo que la exploración de rutas representa una mayor porporción en el uso de los recursos.

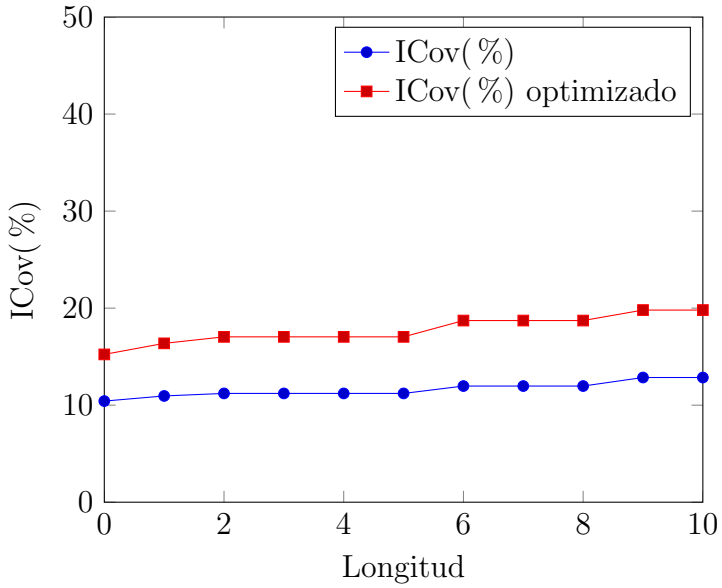


Figura 6.1: Cobertura de instrucciones LLVM

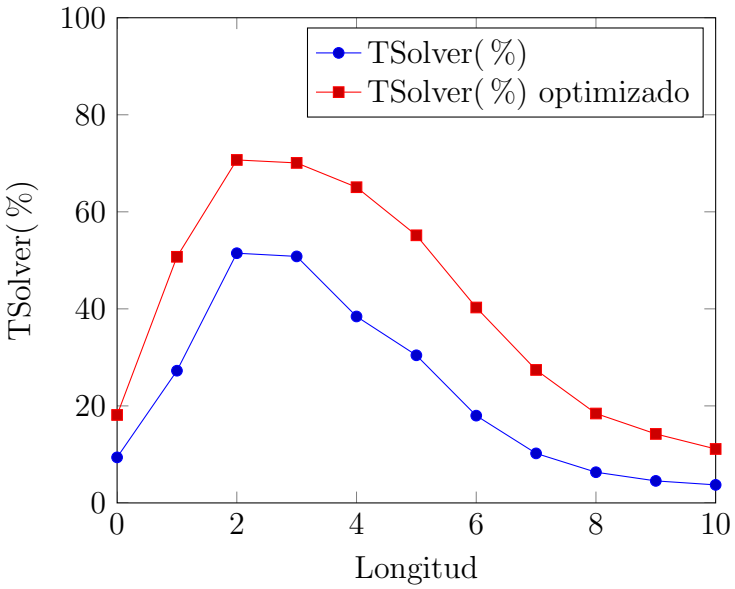


Figura 6.2: Uso del solucionador SMT

Finalmente, se puede comparar estos resultados con los dados en la página de KLEE sobre el mismo test, para el caso de $L = 3$. Los resultados obtenidos son sensiblemente más bajos que los dados en la página. Por ejemplo, se obtiene un

11,21 % y 17,04 % para $ICov(\%)$ frente a los indicados en la página de 22,7 % y 30,16 %. Y también, con un menor uso del solucionador SMT, un 50,81 % y 70,08 % frente a los valores de la página de 62,97 % y 80,66 %. Es difícil saber el origen de estas discrepancias, la versión de Coreutils indicada por la página es anterior a la utilizada en este test y esta puede ser una posible explicación. Tambíen, es cierto que son distintas las versiones tanto de KLEE, como de Linux, con lo que reproducir el test en las mismas condiciones no es factible.

En resumen, los resultados muestran el comportamiento esperado de la exploración frente al tamaño del espacio de búsqueda, aunque este comportamiento es, en términos absolutos, peor al mostrado en la página de KLEE.

Capítulo 7

Conclusiones

Una primera conclusión del estudio realizado es que la búsqueda de vulnerabilidades de programas puede reformularse formalmente como un problema de verificación de programas. Del mismo modo, la búsqueda y síntesis de *exploits* puede estudiarse desde el mismo punto de vista. Cierto es, que no es la única forma de abordar ambos problemas, pero sin duda, los avances que se vayan produciendo en el campo de la verificación contribuirán significativamente a la mejora de las herramientas de análisis de vulnerabilidades.

El avance conseguido en los solucionadores SMT ha revivido la ejecución simbólica como una técnica de aplicación práctica real en el análisis binario de programas, siendo la detección de vulnerabilidades una de sus áreas de aplicación. Sin embargo, aún es necesario un mayor avance de los solucionadores SMT en este área para asentar definitivamente la ejecución simbólica como herramienta de análisis. Se necesita, por ejemplo, su ampliación al tratamiento de la aritmética en punto flotante y de punteros.

En este trabajo, se ha llevado a cabo, también, una exposición de la ejecución simbólica, sus variantes, limitaciones y principales herramientas existentes. Se trata de un campo de gran flexibilidad y abierto a continuas innovaciones.

La creación de motor de ejecuciíon simbólica es un trabajo complejo, que necesita de conocimientos amplios de teoría de lenguajes de programación y de ingente cantidad de código fuente. Razón por la que se ha escogido el motor de ejecución simbólica KLEE, disponible en código abierto, del que se ha hecho una presentación

exhaustiva. KLEE facilita la investigación ya que permite su extensión, y con ello la prueba de variaciones sobre su algoritmo base.

La incorporación de herramientas prácticas de búsqueda de vulnerabilidades como una etapa más en el desarrollo de software supone una mejora considerable en la calidad final de este. Se ha visto que la implementaciíon de una herramienta de estas características a partir de KLEE es posible. Sin embargo, existen características necesarias en una herramienta práctica de test de las que no dispone actualmente KLEE. En este trabajo se han apuntado algunas de esas características, como puede ser la inclusión de *mocks* y la adecuación a un *framework* de test. También, se ha hecho modificaciones de KLEE para comenzar a dotarle de ellas. Aunque, ni las modificaciones ni las características aportadas son todavía suficientes para conseguir una herramienta práctica de aplicación real.

Otro aspecto tratado, es el hecho de que se puede combinar una herramienta como KLEE con otras técnicas de análisis, en este caso un sencillo analizador estático. La intuición parece indicar que la creación de una herramienta potente de análisis binario conlleva la orquestación de varias técnicas.

Por otro lado, los problemas encontrados durante el trabajo realizado parecen constatar que KLEE:

- Se apoya sobre librerías actualmente desactualizadas, su instalación a partir del código fuente es compleja y no está bien documentada.
- Su código no parece tener la calidad necesaria. La prueba ha sido la necesidad de usar la instrucción "*fork()*" en este trabajo, puesto que no se libera bien la memoria al final del proceso principal de ejecución.
- Existe una limitada modelización de la interacción con el entorno.

Lo anterior parece indicar que KLEE no está actualmente preparado para ser una base adecuada de una herramienta comercial.

En Avgerinos et al. (2012) [8] se indica que uno de los mayores problemas de la generación de *exploits*, es que la exploración del espacio de estados del programa sea suficiente. Algo directamente aplicable también a la búsqueda de vulnerabilidades y que ha podido corroborarse en este trabajo. Además, en [8] tampoco se considera

KLEE como un motor de ejecución simbólica completamente satisfactorio, tal vez, por ese motivo crearon MAYHEM.

Capítulo 8

Líneas de trabajo futuras

La búsqueda de vulnerabilides no es un problema definitivamente resuelto y la ejecución simbólica es una técnica que no ha agotado todas sus posibilidades.

Actualmente la aplicación práctica de la ejecucón simbólica fuera del ámbito de la investigación está muy limitado, por lo que una línea de mejora es la creación de herramientas prácticas que posean estabilidad, facilidad de integración y amplias capacidades de modelado y de interacción con el entorno. Estas herramientas necesitan incorporar:

- Implementación del acceso a bases de datos, servicios web como por ejemplo RESTful.
- Modelado de *sockets*, llamadas al sistema y a ensamblador.
- La posibilidad de uso de nuevos solucionadores SMT.

Posiblemente esta línea de trabajo carezca de interés desde un punto de vista de la investigación, pero es absolutamente necesaria en la aplicación práctica.

Por otro lado, existen otras líneas más en el plano teórico o de prueba de concepto:

- Mejoras en los solucionadores SMT, por ejemplo en el campo de la aritmética de coma flotante.
- Ampliación del análisis por ejecución simbólica de los binarios. Actualmente el programa a analizar necesita estar en LLVM IR, sería necesario la posibilidad

de conversión de binario a LLVM IR, explorando las posibilidades del proyecto McSema [22] por ejemplo.

- La posibilidad de combinar análisis estático con el fin de guiar la exploración simbólica.
- La combinación de análisis tintado y ejecución simbólica.

Finalmente, aunque en este trabajo se ha analizado una herramienta orientada al lenguaje C, el lenguaje más estudiado, existen otros lenguajes ampliamente utilizados como Java donde la investigación realizada no es tan amplia.

Bibliografía

[1] T. N. Brooks, "Survey of Automated Vulnerability Detection and Exploit Generation Techniques in Cyber Reasoning Systems," *Computer Research Repository*, vol. abs/1702.06162, 2017.

[2] J. R. Adrados-Pedroche, "Diseño de un analizador de vulnerabilidades en la codificación en C," Master's thesis, Universidad Nacional de Educación a Distancia (UNED), Madrid, 2014.

[3] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Proceedings of the 4th International Conference on Information Systems Security*, (Hyderabad, India), Dec. 2008.

[4] D. Brumley, H. Wang, S. Jha, and D. Song, "Creating Vulnerability Signatures Using Weakest Pre-conditions," 2007.

[5] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit Hardening Made Easy," in *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, 2011.

[6] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic Exploit Generation," in *Network and Distributed System Security Symposium*, February 2011.

[7] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," November 2008.

[8] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on Binary Code," in *IEEE Symposium on Security and Privacy*, pp. 380–394, IEEE Computer Society, 2012.

[9] M. Ruffell, J. B. Hong, H. Kim, and D. S. Kim, "Towards Automated Exploit Generation for Embedded Systems," in *Information Security Applications* (D. Choi and S. Guilley, eds.), pp. 161–173, Springer International Publishing, 2017.

[10] A. Höfler, "SMT solver comparison," in *Diploma seminar*, 2014.

[11] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, pp. 385–394, 1976.

[12] Q. Yi, Z. Yang, S. Guo, C. Wang, J. Liu, and C. Zhao, "Eliminating Path Redundancy via Postconditioned Symbolic Execution," *IEEE Trans. Software Eng.*, vol. 44, no. 1, pp. 25–43, 2018.

[13] C. G. do Val, "Conflict-Driven Symbolic Execution: How to Learn to Get Better," Master's thesis, The University of British Columbia, Canada, 2014.

[14] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic Execution for Software Testing in Practice: Preliminary Assessment," in *ICSE*, pp. 1066–1071, ACM, 2011.

[15] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, pp. 10:1–10:38, 2008.

[16] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems," in *16th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Newport Beach, CA*, ASPLOS, 2011.

[17] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Test for Complex Systems Programs," in *OSDI'08*, pp. 209–224, USENIX Association Berkeley, CA, USA, 2008.

[18] U. of Illinois, "KLEE LLVM Execution Engine." `https://klee.github.io/`, 2017).

[19] Stackoverflow, "Limits of Klee (the LLVM program analysis tool)." `https://stackoverflow.com/questions/5742618/limits-of-klee-the-llvm-program-analysis-tool`, 2015.

[20] T. U. of Munich, "How to build KLEE." `https://github.com/tum-i22/klee-install`, 2018).

[21] I. A. Mason *et al.*, "Whole Program LLVM." `https://github.com/travitch/whole-program-llvm`, 2018).

[22] T. of Bits, "Mcsema, x86 to machine code translation framework." `https://github.com/trailofbits/mcsema`, 2018).

Lista de siglas y acrónimos

ABI	Aplication Binary Interface
AFL	American Fuzzy Loop
API	Application Program Interface
AEG	Automatic Exploit Generation
APEG	Automatic Patch-based Exploit Generation
ASRL	Address Space Layout Randomisation
BCA	Byte-Code Analysis
BSD	Berkeley Software Distribution
CEC	Concrete Executor Client
CFG	Control Flux Graph
CGC	Cyber Grand Challenge
CIL	C Intermediate Language
CMU	Carnegie Mellon University
CNF	Conjunctive Normal Form
CVC	Cooperating Validy Checker
CUTE	Concolic Unit Testing Engine
DARPA	Defense Advanced Research Projects Agency
DART	Directed Automated Random Testing
DBA	Dynamic Binary Analysis
DBT	Dynamic Binary Translator
DEP	Data Execution Prevention
DPLL(T)	Davis-Putman-Logemann-Loveland modulo Theories
EGT	Execution-generated Testing
EXE	Execution Generated Executions

FIFO	First-In, First-Out
FOL	First Order Logic
JIT	Just In Time
JPF-SE	Java PathFinder - Symbolic Execution
GSE	Generalized Symbolic Execution
GPL	GNU General Public License
INRIA	Institut National de Recherche en Informatique et Automatique
LGPL	GNU Lesser General Public License
LLVM	Low Level Virtual Machine (originalmente)
LLVM IR	LLVM Intermediate Representation
MSR-LA	Microsoft Research License Agreement
NLS	National Language Support
POSIX	Portable Operating System Interface
QEMU	Quick Emulator
RESTful	Representational State Transfer (servicios web)
S2E	Selective Symbolic Execution
SAGE	Scalable Automated Guided Execution
SAT	Boolean Satisfiability
SCA	Source Code Analysis
SES	Symbolic Executor Server
SMT	Satisfiability Modulo Theories
SRI	Standford Research Institute
STP	Simple Theorem Prover
UIUC license	University of Illinois/NCSA Open Source License
WLLVM	Whole Program LLVM

Apéndice A

Código fuente librería de utilidades

En este anexo se incluye el código fuente de la librería de apoyo a la escritura de funciones de test. Esta librería se compone de los siguientes *scripts* en Python 3.

Listado A.1: Variables de entorno, *setenv.sh*

```
#/bin/sh
#-----------------------------------------------------------------
# setenv.sh
#
# Symbolic tools, environment variables and alias
# Copyleft (C) 2018, Juan A. Oses
#-----------------------------------------------------------------

# KLEE
export KLEE_ROOT=/home/klee/socle
export KLEE_HOME=$KLEE_ROOT/klee

# LLVM_COMPILER
export LLVM_COMPILER=clang
export LLVM_COMPILER_HOME=$KLEE_ROOT/llvm
export LLVM_COMPILER_PATH=$LLVM_COMPILER_HOME/Release/bin

# U-KLEE
export UKLEE_ROOT=$KLEE_HOME/Release+Asserts
export UKLEE_HOME=$UKLEE_ROOT/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$UKLEE_ROOT/lib

# KLEE alias
alias        klee="$KLEE_HOME/Release+Asserts/bin/klee"
alias     klee-ut="$KLEE_HOME/Release+Asserts/bin/klee-ut"
alias  ktest-tool="$KLEE_HOME/Release+Asserts/bin/ktest-tool"
```

```
alias  klee-stats="$KLEE_HOME/Release+Asserts/bin/klee-stats"
alias  klee-replay="$KLEE_HOME/Release+Asserts/bin/klee-replay"
alias        clang="$LLVM_COMPILER_HOME/Release/bin/clang"
alias          lli="$LLVM_COMPILER_HOME/Release/bin/lli"
alias     llvm-dis="$LLVM_COMPILER_HOME/Release/bin/llvm-dis"
alias  llvm-ranlib="$LLVM_COMPILER_HOME/Release/bin/llvm-ranlib"

# SYMTOOLS
export PATH=$SYM_TOOLS_HOME:$PATH

# SYMTOOL alias
alias symtool="$SYM_TOOLS_HOME/symtool.py"
alias set-mock="export MOCK_SET=true"
alias unset-mock="unset MOCK_SET"
alias make-suite="$SYM_TOOLS_HOME/make_suite.py"
alias set-optimize="export UKLEE_OPTIMIZE=true"
alias unset-optimize="unset UKLEE_OPTIMIZE"
alias simple-analysis="$SYM_TOOLS_HOME/simple_analysis.py"

# Version
export SYMTOOL_VERSION="1.0"
```

Listado A.2: Envoltorio sobre klee-ut, *symtool.py*

```
#!/usr/bin/python
#-----------------------------------------------------------------
# symtool.py
#
# Simple wrapper on Klee-ut
# Copyleft (C) 2018, Juan A. Oses
#-----------------------------------------------------------------
import sys
import os
import subprocess

VERSION=os.environ.get('SYMTOOL_VERSION')
UKLEE_HOME=os.environ.get('UKLEE_HOME')
OUTPUT_PATH="./uklee-out"
UKLEE_BIN="%s/klee-ut" % (UKLEE_HOME)

# Help messages
def help():
    print(" \nsymtool: A Python wrapper on Klee-ut - v%s" % (VERSION))
    print(" \nUSAGE: symtool <input bytecode file.bc>\n")

# Perform main task
def mainTask(*args):
    sym_args = ' '.join(str(x) for x in args)
```

```
    sym_args = sym_args.replace('\'', '').replace('[', '').replace(']', '').replace(
        ',', '')
    optimize = os.environ.get('UKLEE_OPTIMIZE')
    if optimize == 'true':
        arguments = "%s --optimize --libc=uclibc --posix-runtime %s" % (UKLEE_BIN,
            sym_args)
    else:
        arguments = "%s --libc=uclibc --posix-runtime %s" % (UKLEE_BIN, sym_args)

    print("Command: %s" % (arguments))
    env = os.environ.copy()
    subprocess.call(arguments, shell=True, env=env)

# Remplacement for the main function
if __name__ == "__main__":
    if len(sys.argv) > 1:
        arg1 = sys.argv[1]
        if arg1 == '--help' or arg1 == '-h':
            help()
            sys.exit(0)
        else:
            if arg1.startswith('-'):
                print("[ERROR]: Unknown option")
                help()
                sys.exit(0)

        if os.path.exists(OUTPUT_PATH):
            shutil.rmtree(OUTPUT_PATH)

        sys.argv.pop(0)
        mainTask(sys.argv)

    else:
        help()
```

Listado A.3: Analizador estático, *simple_analysis.py*

```
#!/usr/bin/python
#-----------------------------------------------------------------
# simple-analysis.py
#
# Simple static analysis for LLVM IR code, looking for
# the number of program arguments and maximum size allocation.
#
# Copyleft (C) 2018, Juan A. Oses
#-----------------------------------------------------------------
import sys
import os
```

```
import re
import subprocess

DEBUG_LEVEL=1
VERSION=os.environ.get('SYMTOOL_VERSION')
LLVM_PATH=os.environ.get('LLVM_COMPILER_PATH')
LLVM_DIS="%s/llvm-dis" % (LLVM_PATH)

# Print debug messages
def debug (str, level=1):
    if level < DEBUG_LEVEL:
        print("[DBG] " + str)

# Help messages
def help():
    print(" \nsymple-analysis: Simple static analyis - v%s" % (VERSION))
    print(" \nUSAGE: simple-analysis <input bytecode file.bc>\n")

# Delete file
def deleteFile(filename):
    try:
        os.remove(filename)
    except OSError:
        pass

# Find the maximum allocation size
def findMaxAllocation(filename):
    maxAlloc = 0
    debug("Looking for the maximum allocation size in %s" % (filename), 1)
    with open(filename, "r") as f:
        for line in f:
            match = re.search('^.*alloca .*, align .*$', line)
            if match:
                debug("line alloca %s:" % (line), 2)
                match = re.search('^.*alloca \[.*$', line)
                if match:
                   sub_line = line.split(',')[0].strip().split('x')[0].strip().split
                       ('[')[1].strip()
                   allocSize = sub_line
                  allocSize = sub_line
                else:
                   allocSize = line.split(',')[1].strip().split(' ')[1]

                if allocSize and int(allocSize) > maxAlloc:
                    maxAlloc = int(allocSize)

            match = re.search('^.*call.*malloc.*$', line)
            if match:
                subLine = line.split(',')[0]
```

```
                allocSize = subLine[subLine.index('(') : subLine.index(')')].split('
                    ')[1]
                if allocSize.isdigit() and allocSize and int(allocSize) > maxAlloc:
                    maxAlloc = int(allocSize)

    debug("Maximum allocation: %d" % (maxAlloc), 1)
    return maxAlloc

# Find the variable containing argv
def findArgvVar(filename):
    isMainFunc = False
    maxArg = 0
    token = None

    debug("Looking for the variable which contains argument pointer in %s" % (
        filename), 2)
    with open(filename, "r") as f:
        for line in f:
            if isMainFunc:
                match = re.search('^.*store[\s].*\%argv,.*$', line)
                if match:
                    token = line.strip().split(',')[1].strip().split(' ')[1].strip()
            else:
                isMainFunc = re.search('^define.*@main.*$', line)

    debug("Variable argv: %s" % (token), 2)
    return token

# Find maximum number of arguments
def findNumArgs(filename):
    tokenList = []
    isMainFunc = False
    maxArg = 0

    debug("Looking for the number of program arguments on %s" % (filename), 1)
    var_argv = findArgvVar(filename)
    with open(filename, "r") as f:
        for line in f:
            if isMainFunc:
                aux = '^.*load.*[\s]%'
                            loadElement = '%s%s.*$' % (aux, var_argv.replace("%"
                                ,""))
                match = re.search(loadElement, line)
                if match:
                    token = line.strip().split(' ')[0]
                    auxToken = '^.*= getelementptr inbounds.*[\s]%'
                    tokenElement = '%s%s,.*$' % (auxToken, token.replace("%", ""))
                    tokenList.append(tokenElement)
            else:
```

```
                isMainFunc = re.search('^define.*@main.*$', line)

    isMainFunc = False
    with open(filename, "r") as f:
        for line in f:
            if isMainFunc:
                for x in tokenList[:]:
                    match = re.search(x, line)
                    if match:
                        numArg = line.split(',')[1].strip().split(' ')[1]
                        if numArg.startswith("%"):
                            print("Index in variable found: %s" % (numArg))
                        else:
                            if numArg and int(numArg) > maxArg:
                                maxArg = int(numArg)
            else:
                isMainFunc = re.search('^define.*@main.*$', line)

    debug("Maximum number of program arguments: %d" % (maxArg), 1)
    return maxArg

# Perform main task
def mainTask(file_name):
    prog_name = file_name.split('/')[-1]
    tempFile = "%s.ll" % (prog_name)
    arguments = "%s %s -o %s" % (LLVM_DIS, file_name, tempFile)

    env = os.environ.copy()
    process = subprocess.call(arguments, shell=True, env=env)

    maxAlloc = findMaxAllocation(tempFile)
    if maxAlloc > 0:
        maxAllocS = round(1.1*maxAlloc)
    else:
        maxAllocS = 0

    maxNumArgs = findNumArgs(tempFile)
    print("Maximum number of arguments: %d" % (maxNumArgs))
    print("Maximum size of allocation: %d" % (maxAlloc))
    print("Maximum buffer size suggested: %d" % (maxAllocS))
    deleteFile(tempFile)

# Remplacement for the main function
if __name__ == "__main__":
    if len(sys.argv) > 1:
        arg1 = sys.argv[1]
        if arg1 == '--help' or arg1 == '-h':
            help()
            sys.exit(0)
```

```
        else:
            if arg1.startswith('-'):
                print("[ERROR]: Unknown option")
                help()
                sys.exit(0)

        filename = sys.argv[1]
        mainTask(filename)

    else:
        help()
```

Listado A.4: Generador fichero de funciones de test, *make_suite.py*

```
#!/usr/bin/python
#---------------------------------------------------------------------
# make_suite.py
#
# Generation of a test suite from the input test source code.
# Result is written in a text file .ts
#
# Copyleft (C) 2018, Juan A. Oses
#---------------------------------------------------------------------
import sys
import re

# Write test suite to file
def writeTestSuite(filename, listFunctions):
    dot_index = filename.index('.')
    fname = filename[0:dot_index]
    fname = fname + ".ts"
    print("Generating %s ...\n" % (fname))
    f = open(fname, 'w')
    for item in listFunctions:
        strItem = str(item).replace('[', '').replace(']', '').replace('\'', '')
        f.write("%s\n" % (strItem))
    f.close()
    print("done!")

# Perform main task
def mainTask(filename):
    functionList = []
    print("Looking for test functions in %s" % (filename))
    with open(filename, "r") as f:
        for line in f:
            match = re.search('^.*Begin.Test.*$', line)
            if match:
                begin_index = line.index("(")
```

```
                end_index = line.index(")", begin_index)
                test_function = line[begin_index:end_index]
                test_function = test_function.replace('(', 'test_');
                functionList.append([test_function])
    return functionList

# Remplacement for the main function
if __name__ == "__main__":
    if len(sys.argv) > 1:
        filename = sys.argv[1]
        fList = mainTask(filename)
        if fList:
            writeTestSuite(filename, fList)
        else:
            print("No test functions found!\n")
    else:
        print("File name is absent!\n")
```

Otra parte de la librería se compone de archivos en C, en el subdirectorio "*include*" : un "*header*" y un archivo para incluir en el código.

Listado A.5: Declaración de funciones y macros, *sym_tool.h*

```
/************************************************
* sym_tools.h
*
* Auxiliary function declaration and macros.
* Copyleft (C) 2018, Juan A. Oses
************************************************/
#ifndef __SYM_TOOLS_H__
#define __SYM_TOOLS_H__

#ifdef __cplusplus
extern "C" {
#endif

int isBasicNumericChar(int, char*);

#define ASSUME_NUMERIC(size, arr)  klee_assume(isBasicNumericChar(size, arr)); \
                                   klee_assume(arr[size - 1] == '\0');

#define _Begin_Test(NAME)  int test_##NAME(int argc, char *argv[]){
#define _End_Test          return 0;}

#ifdef __cplusplus
}
#endif
```

```
#endif /* __SYM_TOOLS_H__ */
```

Listado A.6: Fragmento a incluir mediante "#include", *sym_tool.inc*

```
/************************************************
* sym_tools.inc
*
* Auxiliary function to check a numeric string.
* Copyleft (C) 2018, Juan A. Oses
************************************************/

#ifndef __SYM_TOOLS_INC__
#define __SYM_TOOLS_INC__

/*
 * Check whether characters of a string are digits or '-'
 */
int isBasicNumericChar(int size, char *str) {
  int k, res = 1;
  char c;

  for (k = 0; k < size - 1; k++) {
    c = str[k];
    res &= (c == '0') ||  (c == '1') || (c  == '2') || (c == '3')
              || (c == '4') || (c == '5') || (c == '6')
              || (c == '7') || (c == '8') || (c == '9') || (c == '-')
              || (c == '.') || (c == '+');
  }
  return res;
}

#endif /* __SYM_TOOLS_INC__ */
```

Apéndice B

Modificaciones código de KLEE

En este apéndice se incluyen las modificaciones hechas en el código fuente de KLEE para crear la utilidad "**klee-ut**".

En el directorio "*klee/tools/klee-ut*", las modificaciones de "*main.cpp*" respecto a "*klee*" son:

Listado B.1: Modificaciones sobre *main.cpp*

```
...
#include "klee/klee-ut.h"
...

#ifndef SUPPORT_KLEE_UCLIBC
static llvm::Module *changeMainFunction(llvm::Module *mainModule, StringRef
    functionName) {
  klee_error("invalid libc, no uclibc support!\n");
}
#else

static llvm::Module *changeMainFunction(llvm::Module *mainModule, StringRef
    functionName) {
  Function *userMainFn = mainModule->getFunction(EntryPoint);
  assert(userMainFn && "unable to get user main");

  cl::opt<std::string>
  SwapEntryPoint("swap-entry-point",
                 cl::desc("Consider the function with the given name as the entrypoint
                     "),
                 cl::init(functionName.data()));

  Function *userTestFn = mainModule->getFunction(SwapEntryPoint);
  //assert(userTestFn && "unable to get test function");
```

```
  if ((userMainFn != NULL) && (userTestFn != NULL)) {
    klee_message("Replacing %s function with main", functionName.data());
    userMainFn->setName("__true_user_main");
    userTestFn->setName("main");
  }

  return mainModule;
}
...
#endif

...

pTList detectTestFunction(int argc, char **argv) {
  int k;
  pTList pList = NULL;

  for (k = 0; k < argc; k++) {
    if ((strcmp(argv[k], "-run-function") == 0) || (strcmp(argv[k], "--run-function"
        ) == 0)) {
      if (++k == argc) {
         klee_message("--run-function expects <function-name>");
         break;
      }
      pList = init_stack();
      push_element(pList, argv[k++]);
      break;
    } else if ((strcmp(argv[k], "-run-test-suite") == 0) || (strcmp(argv[k], "--run-
        test-suite") == 0)) {
      if (++k == argc) {
         klee_message("--run-test-suite expects <test-suite-file>");
         break;
      }

      pList = get_test_suite(argv[k++]);
      dump_stack_suite(pList);
    }
  }
  return pList;
}

/* KLEE original main function */
int main_orig(int argc, char **argv, char **envp, char *testFunction) {
  ...
  case UcLibc:
    if (testFunction != NULL) {
       std::string NameTestFunction = std::string(testFunction);
       mainModule = changeMainFunction(mainModule, NameTestFunction);
```

```
    }

    mainModule = linkWithUclibc(mainModule, LibraryDir);
    break;
  }
  ...
 }

/* KLEE-UT main function */
int main(int argc, char **argv, char **envp) {
  int i, res, taille = 0;
  pid_t child_pid, wpid;
  int status = 0;
  char buffer[BUFFER_SIZE];

  pTList pList = detectTestFunction(argc, argv);
  if (pList == NULL) {
    return main_orig(argc, argv, envp, NULL);
  }

  taille = pList->size;
  for (i=0; i<taille; i++) {
    pop_element(pList, buffer, &res);
    printf("Test: %s\n", buffer);
    if (res < 0) {
      continue;
    }
    if ((child_pid = fork()) == 0) {
      main_orig(argc, argv, envp, buffer);
      exit(0);
    }
    while ((wpid = wait(&status)) > 0);
  }
  return 0;
}
```

En el archivo "*Makefile*" del mismo directorio basta con indicar: "*TOOLNAME = klee-ut*". Y el archivo "*CMakeLists.txt*" queda:

Listado B.2: Fichero *CMakeLists.txt*

```
add_executable(klee-ut
  main.cpp
)

set(KLEE_LIBS
  kleeCore
)
```

```
target_link_libraries(klee-ut ${KLEE_LIBS})

install(TARGETS klee-ut RUNTIME DESTINATION bin)

# The KLEE binary depends on the runtimes
add_dependencies(klee-ut BuildKLEERuntimes)
```

Por otro lado, la implementación de funciones auxiliares se hace en el archivo "*KleeUt.c*", directorio "*klee/lib/Core*":

Listado B.3: Implementación de una lista, *KleeUt.c*

```
-/*****************************************************
* KleeUt.c
*
* Auxiliary code to manage a list.
* Copyleft (C) 2018, Juan A. Oses
*****************************************************/
#include <stdio.h>
#include <klee/klee-ut.h>

pTList init_stack() {
  pTList pList = (pTList) malloc (sizeof(TList));
  pList->size = 0;
  return pList;
}

pTList push_element(pTList pList, char *data) {
  size_t len = strlen(data);
  pTCell cell = (pTCell) malloc (sizeof(TCell));
  if (cell != NULL) {
    cell->data = (char*) malloc(len * sizeof(char));
        if (cell->data != NULL) {
           strcpy(cell->data, data);
           cell->next = NULL;
           if (pList->size != 0) {
               pList->tail->next = cell;
           } else {
               pList->cells = cell;
           }
           pList->tail = cell;
           pList->size++;
        }
  }
  return pList;
}

pTList pop_element(pTList pList, char *value, int *result) {
    pTCell cell;
```

```
    *result = -1;
    if (pList == NULL) {
        return NULL;
    }

  strcpy(value, pList->cells->data);
  cell = pList->cells->next;
  free(pList->cells);

  pList->cells = cell;
  pList->size--;
  *result = 0;
  return pList;
}

void dump_stack_suite(pTList stack) {
  char buffer[BUFFER_SIZE];
  int i, taille, result;

  if (stack == NULL) {
    printf("Stack is Null!\n");
    return;
  }

  taille = stack->size;
  printf("Stack size: %d\n", (unsigned short) taille);
  for (i=0; i < taille; i++) {
    result = read_element(stack, buffer, i);
    if (result > -1) {
      printf("value at %d : %s \n", i, buffer);
    } else {
      printf("Error reading %d element!\n", i);
    }
  }
}

pTList get_test_suite(char *filename) {
  FILE *fptr;
  char *c, line[BUFFER_SIZE];
  pTList stack = init_stack();

  if ((fptr = fopen(filename, "r")) == NULL) {
    printf("Error! opening file!\n");
    return NULL;
  }

  while (fgets(line, BUFFER_SIZE, fptr) != NULL) {
    c = strchr(line, '\n');
    if (c) {
```

```
      *c = 0;
    }
    push_element(stack, line);
  }
  fclose(fptr);
  return stack;
}

int read_element(pTList pTlist, char *buffer, int pos) {
  int i;
  pTCell pCell;

  if (pTlist == NULL) {
        return -1;
  }

  if (pos > (unsigned short) pTlist->size - 1) {
        return -1;
  }

  pCell = pTlist->cells;
  for (i = 0; i < pos; i++) {
        pCell = pCell -> next;
  }

  strcpy(buffer, pCell->data);
  return 0;
}
```

Y finalmente, el "*header*" se incluye en el directorio "*klee/include/klee*".

Listado B.4: Declaración de una lista dinámica *klee-ut.h*

```
/*******************************************************
* klee-ut.h
*
* Declaration of auxiliary code to manage a list.
* Copyleft (C) 2018, Juan A. Oses
*******************************************************/
#ifndef __KLEE_UT_H__
#define __KLEE_UT_H__

#include "string.h"
#include "stdlib.h"

#define BUFFER_SIZE 1024

#ifdef __cplusplus
extern "C" {
#endif
```

```
typedef struct TCell {
    char *data;
    struct TCell *next;
} TCell;

typedef TCell* pTCell;

typedef struct TList {
   pTCell cells;
   pTCell tail;
   size_t size;
} TList;

typedef TList* pTList;

typedef pTList (*pTest_Suite) (void);

pTList init_stack();
pTList push_element(pTList pList, char *data);
pTList pop_element(pTList pList, char *value, int *result);
pTList get_test_suite(char *filename);
void dump_stack_suite(pTList stack);
int read_element(pTList pTlist, char *buffer, int pos);

#ifdef __cplusplus
}
#endif
```

Apéndice C

Modificaciones código de klee-uclibc

A continuación, se indican los cambios hechos en la librería "**klee-uclib**". De esta forma, sobre el código fuente de "*syslog.c*", en el directorio "*klee-uclibc/libc/misc/syslog*" los cambios realizados han sido:

Listado C.1: Modificaciones en *syslog.c*

```
...
#include <mock.h>
...

/*
 * OPENLOG -- open system log
 */
void
openlog( const char *ident, int logstat, int logfac )
{
..
retry:
    if (LogStat & LOG_NDELAY) {
       if (IS_MOCK()) {
          print_trace_mock("openlog");
          LogFile = 0;
       } else {
          if ((LogFile = socket(AF_UNIX, logType, 0)) == -1) {
            goto DONE;
          }
       }
    }
    ...
```

```
    if (LogFile != -1 && !connected) {
        if (IS_MOCK()) {
          print_trace_mock("openlog, connected = 1");
          connected = 1;
        } else {
          if (connect(LogFile, &SyslogAddr, sizeof(SyslogAddr) -
                    sizeof(SyslogAddr.sa_data) + strlen(SyslogAddr.sa_data)) != -1)
          {
          ...
          }
        }
...
}

...
void
syslog(int pri, const char *fmt, ...)
{
        va_list ap;

        va_start(ap, fmt);
        if (IS_MOCK()) {
          print_trace_mock("syslog");
        } else {
          vsyslog(pri, fmt, ap);
        }
        va_end(ap);
}
```

En el directorio "*klee-uclibc/include*" se incluye el archivo "*header*" siguiente:

Listado C.2: Declaración para funcionalidad mock, *mock.h*

```
/*****************************************************
* mock.h
*
* Declarations for mock code.
* Copyleft (C) 2018, Juan A. Oses
*****************************************************/
#ifndef _MOCK_H
#define _MOCK_H

#ifdef __cplusplus
extern "C" {
#endif

#define MOCK_SET "MOCK_SET"

#define IS_MOCK(x) is_mock(x) == 0
```

```
void print_trace_mock(char *name);
int is_mock();

#ifdef __cplusplus
} /* end of extern "C" */
#endif

#endif
```

Por otro lado, en el directorio "*klee-uclibc/libc/misc/mock*" se incluyen los siguientes archivos.

Listado C.3: Funcionalidades *mock*, *mock.c*

```
/*************************************************
* mock.c
*
* Simple mock code.
* Copyleft (C) 2018, Juan A. Oses
*************************************************/
#include <mock.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define BUFFER_SIZE 100

void print_trace_mock(char *name) {
  char *isMock = getenv(MOCK_SET);
  char trace[BUFFER_SIZE];
  size_t len;

  memset(trace, 0, BUFFER_SIZE*sizeof(char));
  if (isMock == NULL) {
    strcpy(trace, "false");
  } else {
    len = strlen(isMock);
    if (len < BUFFER_SIZE) strcpy(trace, isMock);
    else strncpy(trace, isMock, (BUFFER_SIZE - 1)*sizeof(char));
  }
  printf("Mock at %s is %s\n", name, trace);
}

int is_mock() {
  char *isMock = getenv(MOCK_SET);

  if (isMock == NULL) {
    return -1;
```

```
  }
  return strcmp(isMock, "true");
}
```

Listado C.4: Fichero *Makefile*

```
top_srcdir=../../../
top_builddir=../../../
all: objs
include $(top_builddir)Rules.mak
include Makefile.in
include $(top_srcdir)Makerules
```

Listado C.5: Fichero *Makefile.in*

```
CSRC := mock.c

MISC_MOCK_DIR := $(top_srcdir)libc/misc/mock
MISC_MOCK_OUT := $(top_builddir)libc/misc/mock

MISC_MOCK_SRC := $(patsubst %.c,$(MISC_MOCK_DIR)/%.c,$(CSRC))
MISC_MOCK_OBJ := $(patsubst %.c,$(MISC_MOCK_OUT)/%.o,$(CSRC))

libc-y += $(MISC_MOCK_OBJ)

objclean-y += misc_mock_objclean

misc_mock_objclean:
        $(RM) $(MISC_MOCK_OUT)/*.{o,os}
```