Verificador de programas eBPF simplificado

Ejercicio N° 2

Objetivos	● Diseño y construcción de sistemas orientados a objetos ● Diseño y construcción de sistemas con procesamiento concurrente ● Protección de los recursos compartidos
Instancias de Entrega	**Entrega 1:** clase 6 (03/11/2020). **Entrega 2:** clase 8 (17/11/2020).
Temas de Repaso	● Threads en C++11 ● Clases en C++11
Criterios de Evaluación	● Criterios de ejercicios anteriores ● Orientación a objetos del sistema ● Empleo de estructuras comunes C++ (string, fstreams, etc) en reemplazo de su contrapartida en C (char*, FILE*, etc) ● Uso de **const** en la definición de métodos y parámetros ● Empleo de constructores y destructores de forma simétrica ● Buen uso del stack para construcción de objetos automáticos ● Ausencia de condiciones de carrera e interbloqueo en el acceso a recursos ● Buen uso de Mutex, Condition Variables y Monitores para el acceso a recursos compartido

Índice

Introducción

Berkeley Packet Filter o BPF fue una tecnología desarrollada para filtrar paquetes de red e integrada en el kernel de Linux hace varios años atrás.

Aunque eficiente, BPF no tomaba ninguna ventaja de los hardwares modernos.

Fue alrededor del 2014 que se introdujo un rediseño: extended BPF o eBPF. [1]

Con soporte para registros de 64 bits y compilación a código nativo *just-in-time*, eBPF tenía una mejor performance que el clásico BPF (ahora llamado cBPF).

Y no tardó en llegar a user space: los usuarios podían crear sus propios programas eBPF y cargarlos al kernel.

Pero un gran poder conlleva una gran responsabilidad.

Un programa eBPF con bugs o mal intencionado podría corromper al kernel o como mínimo colgar la máquina.

Es aquí donde el verificador de programas eBPF entra en juego.

Implementado en el mismísimo kernel de Linux, el verificador detecta operaciones de memoria inválidos, bucles de código que podrían colgar el sistema entre otras acciones que podrían degradar la performance del sistema o incluso corromperlo.

Descripción

En este trabajo se implementará una versión simplificada del verificador que sólo verificará la existencia o no de bucles de código y/o instrucciones sin ejecutar.

Este es un ejemplo de un código eBPF:

```
            ldh [12]
            jne #0x806, drop
            ret #-1
     drop:  ret #0
```

Cada línea consta de una única instrucción que puede o no estar prefijada por una etiqueta.

Una etiqueta es una palabra que puede tener letras y/o números.

De las 4 líneas mostradas, la última es la única que tiene una etiqueta llamada "`drop`".

Puede verse que las etiquetas siempre van seguidas de un dos puntos ":".

Cada instrucción tiene un único comando u opcode al que le siguen cero o más argumentos. En el caso de haber más argumentos estos se separan por comas "," y un espacio.

Por ejemplo la instrucción `ldh [12]` tiene como comando "`ldh`" y como único argumento el "`[12]`".

Mientras que la instrucción "`jne #0x806, drop`" tiene como comando "`jne`" y dos argumentos: "`#0x806`" y "`drop`".

Como puede verse hay siempre al menos un espacio entre el comando y los argumentos y entre los argumentos.

Las siguientes instrucciones NO son válidas:

```
        ldh[12]
        jne #0x806,drop
        jne #0x806 drop
drop:ret #-1
```

Asimismo habrá siempre un espacio entre la etiqueta, si existiese, y el resto de la instrucción.

El siguiente código NO es válido:

```
drop:
        ret #0
```

Es posible, sin embargo, que haya líneas vacías entre las instrucciones a fin de hacer más fácil la lectura de los programas.

Por ejemplo lo siguiente ES válido:

```
        ldh [12]
        jne #0x806, drop
        ret #-1

drop:   ret #0
```

Se garantiza que todos los programas eBPF serán siempre válidos sintacticamente para simplificar la implementación.

Jumps o instrucciones de salto

Los siguientes comandos u *opcodes* son los correspondientes jumps o saltos. Con ellos un programa eBPF puede construir lo que en C/C++ serían los `if` y los `while`:

```
jmp
ja
jeq
jneq
jne
jlt
jle
jgt
jge
jset
```

Hay tres modos de operación o formas de salto: las incondicionales con un argumento, las condicionales con dos argumentos y las condicionales con tres argumentos.

Los saltos incondicionales son un comando u opcode de jump seguido de un único argumento que es una etiqueta.

Por ejemplo:

```
jmp drop
```

Los saltos incondicionales indican que el programa al ejecutar dicha instrucción salta a la instrucción etiquetada (en el ejemplo hacia "drop") y la ejecución continua desde ahí. En C/C++ sería el equivalente al `goto`.

Los saltos condicionales de dos argumentos evalúan el primer argumento y si la evaluación es true saltan a la instrucción cuya etiqueta es el segundo argumento.

Por ejemplo:

```
jne #0x806, drop
```

La instrucción evalúa "`#0x806`" y si es true salta a "`drop`". Si la evaluación es *false* NO hay un salto y el programa continúa con la siguiente instrucción al jump. En C/C++ sería el equivalente a un `if`.

Los saltos condicionales de tres argumentos son iguales a los de dos argumentos pero reciben dos etiquetas y no una.

Si la evaluación es *true* saltan a la instrucción etiquetada con la primera etiqueta mientras que si es *false* saltan a la segunda. Al igual que los saltos incondicionales y a diferencia de los saltos condicionales de dos argumentos, los saltos de tres argumentos siempre saltan.

Por ejemplo:

```
jeq #1, etiquetatrue, etiquetafalse
```

Como está garantizado que la sintaxis de los programas eBPF es siempre válida podemos resumir lo mencionado en las siguientes tres fórmulas:

```
<jmp-opcode> <etiqueta>
<jmp-opcode> <arg>, <etiqueta>
<jmp-opcode> <arg>, <etiqueta>, <etiqueta>
```

Se garantiza que todas las etiquetas que aparezcan en una instrucción de salto estará definida. Esto es, existirá una única instrucción que tendrá dicha etiqueta.

`ret` o instrucciones de retorno

Las instrucciones "`ret`" pueden tener cero o más argumentos y marcan el fin de un programa eBPF.

Modelado del programa como grafo.

Tomemos una vez más el siguiente ejemplo:

```
        ldh [12]
        jne #0x806, drop
        ret #-1
drop:   ret #0
```

La segunda instrucción es un salto condicional de dos argumentos. Puede saltar a la etiqueta "`drop`" o puede no saltar.

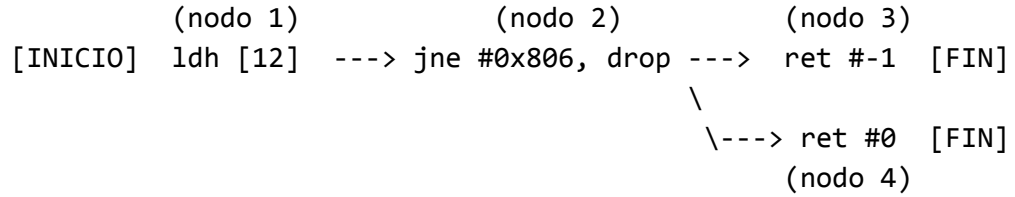
Las tercer y cuarta instrucciones son "`ret`" y finalizan el programa.

Todo programa eBPF arranca con la primera instrucción.

Ahora podemos modelar este programa como un grafo dirigido: los nodos serán las instrucciones y las aristas serán los posibles saltos.

Toda instrucción que no sea un jump o un ret puede modelarse como un nodo con una arista hacia la siguiente instrucción.

Siguiendo el ejemplo nuestro grafo quedaria asi:



Nótese cómo para poder modelar el programa eBPF en un grafo es necesario detectar los jumps, los `ret`, las etiquetas pero NO es necesario tener un parsing completo del programa eBPF y NO es necesario "evaluar" el programa.

Detección de ciclos o bucles

El verificador de eBPF deberá detectar ciclos o bucles en el código. Esto es saltos hacia instrucciones ya ejecutadas (lo que en C/C++ sería un `while`).

Al garantizar la no existencia de ciclos se puede garantizar que el programa de eBPF no se podrá colgar (por ejemplo en un loop infinito).

Si se modela un programa como un grafo dirigido, la detección de ciclos se reduce tan solo a una búsqueda *Deep First Search* o DFS. [2] [3]

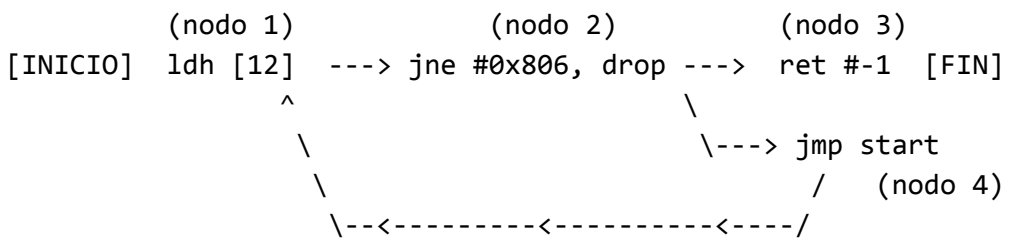
En otras palabras, buscamos ver si el grafo dirigido es acíclico (*Directed Acyclic Graph* o DAG).

DFS es una de las muchas estrategias, pero es sin duda la más simple.

Este es un ejemplo de un programa que tiene un ciclo:

```
start:  ldh [12]
        jne #0x806, drop
        ret #-1
drop:   jmp start
```

Y este sería su grafo asociado:



Detección de instrucciones sin usar

DFS no solo detecta los ciclos sino que además, por la naturaleza del algoritmo, recorre todo el grafo visitando todos los nodos, o sea, todas las instrucciones.

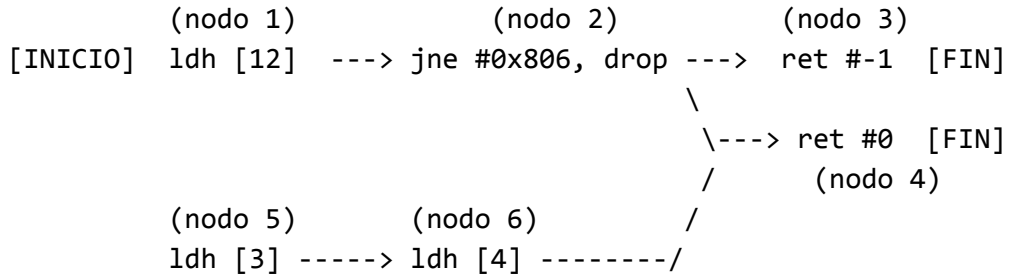
Si alguna instrucción no es visitada durante el DFS es porque no puede ser accedida y por ende es una

instrucción sin usar que nunca sería ejecutada.

Esto es un ejemplo de un programa que tiene dos instrucciones sin usar:

```
          ldh [12]
          jne #0x806, drop
          ret #-1
          ldh [3]
          ldh [4]
drop:     ret #0
```

Y este sería su grafo:



Nótese cómo el algoritmo de DFS, empezando desde el nodo 1 podrá recorrer los nodos 1, 2, 3 y 4 pero no podrá visitar ni al nodo 5 ni el 6.

El grafo no tiene ciclos pero el DFS detecta también que instrucciones no podrán ser ejecutadas nunca.

Verificación

El verificador de programas deberá procesar los archivos pasados por línea de comandos y determinar si tienen ciclos y/o instrucciones sin usar.

Deberá imprimir por pantalla el nombre de cada archivo procesado y un mensaje indicando los resultados linea a linea, con la salida ordenada alfabéticamente.

Ejemplo:

```
./tp 1 arp.bpf not_exec_inst.bpf bad_tcp.bpf
arp.bpf GOOD
bad_tcp.bpf FAIL: cycle detected
not_exec_inst.bpf FAIL: unused instructions detected
```

Nótese el orden de las líneas y los mensajes:

`GOOD` en el caso de no haberse encontrado ciclos ni instrucciones sin uso.
`FAIL: cycle detected` en el caso de haberse detectado un ciclo

`FAIL: unused instructions detected` en el caso de instrucciones sin uso.

Si un programa eBPF tuviese ciclos e instrucciones sin uso, el verificador deberá imprimir solo el mensaje correspondiente al de los ciclos.

Verificación en paralelo

El verificador deberá procesar los archivos en paralelo con una serie de threads definidos por línea de comandos.

El siguiente ejemplo ejecuta el verificador con 2 hilos para procesar 3 archivos:

```
./tp 2 arp.bpf not_exec_inst.bpf bad_tcp.bpf
arp.bpf GOOD
bad_tcp.bpf FAIL: cycle detected
not_exec_inst.bpf FAIL: unused instructions detected
```

Es requerimiento del trabajo práctico que se implemente un objeto que servirá de repositorio de los archivos a ser procesados.

Cada hilo deberá pedir un archivo a este objeto, parsearlo, modelarlo y analizarlo para luego volver a pedir otro archivo y repetir la operación.

Esta restricción del enunciado fuerza a que haya un objeto compartido. Recuerde protegerlo!

Por el otro lado, los hilos deberán guardar los resultados a medida que los vayan obteniendo en un único objeto compartido.

Esta restricción del enunciado fuerza a que haya otro objeto compartido. Recuerde protegerlo también!

Solo cuando todos los archivos hayan sido procesados se imprimirán los resultados.

Archivos de entrada

Los archivos de entrada son programas eBPF sintácticamente correctos como lo descrito anteriormente:

```
        ldh [12]
        jne #0x806, drop
        ret #-1
drop:   ret #0
```

Formato de Línea de Comandos

El programa se ejecutará pasando por línea de comandos recibiendo la cantidad de hilos y luego uno o más archivos:

```
./tp <num hilos> <archivo> [<archivo>...]
```

Códigos de Retorno

El programa retornará 0 siempre.

Entrada y Salida Estándar

La entrada estándar no será utilizada.

La salida estándar de errores es libre, y el alumno puede utilizarla para imprimir mensajes útiles de error, a fin de corroborar alguna teoría al subir la implementación a Sercom en caso de tener problemas que solamente se reproduzcan en ese entorno.

La salida estándar se utilizará al final de la ejecución como se lo describió anteriormente.

Ejemplos de Ejecución

A continuación, se mostrarán algunos ejemplos de ejecución.
Consideremos los siguientes tres archivos:

Archivo `arp.bpf`

```
            ldh [12]
            jne #0x806, drop
            ret #-1
    drop:   ret #0
```

Archivo `not_exec_inst.bpf`

```
            ld [0]
            ld [1]
            ret #-1
            ld [2]
```

Archivo `bad_tcp.bpf`

```
            ldh [12]
    up:     jne #0x800, drop
            ldb [23]
            jneq #6, up
            ret #-1
    drop:   ret #0
```

Los tres archivos se procesaran con 2 hilos de procesamiento y la salida esperada es:

```
./tp 2 arp.bpf not_exec_inst.bpf bad_tcp.bpf
arp.bpf GOOD
bad_tcp.bpf FAIL: cycle detected
not_exec_inst.bpf FAIL: unused instructions detected
```

Restricciones

La siguiente es una lista de restricciones técnicas:

1. El sistema debe desarrollarse en ISO C++11.
2. Está prohibido el uso de variables globales.
3. Deberá haber un **único objeto compartido** que tenga los archivos y serán los hilos de procesamiento que le irán pidiendo un archivo a la vez para procesarlos.
4. Deberá haber un **único objeto compartido** que guarde los resultados a medida que los hilos los vayan generando. Al terminar todos los archivos, se imprimen de ahí los resultados por pantalla.

Recomendaciones

1. Leer las partes en *negrita* del enunciado. Hay varias simplificaciones que se resaltaron porque pueden simplificar la implementación. A tomarlas en cuenta.
2. Entender e implementar la solución al problema pedido. En otras palabras, no implementar algo que no se pidió o que no sirve para resolver el trabajo.
3. Notar e investigar las diferencias entre `lock_guard`, `unique_lock`, y `shared_lock`.
4. Generar casos de prueba propios y hacer un análisis parecido al del enunciado.

Referencias

[1] https://lwn.net/Articles/740157/
[2] https://en.wikipedia.org/wiki/Topological_sorting
[3] https://en.wikipedia.org/wiki/Cycle_(graph_theory)