Ingeniería del Software II

Taller #5 – Greybox Fuzzing

LEER EL ENUNCIADO COMPLETO ANTES DE ARRANCAR.

Fecha de entrega: 2 de Mayo de 2024
Fecha de re-entrega: 23 de Mayo de 2024 (no hay extensiones)

Setup

El Taller provee un archivo `requirements.txt` para instalar todas las dependencias necesarias en un ambiente virtual de Python. En particular, este taller require que utilicemos Python>=3.9 y fuzzingbook >= `1.1`. Para instalar estas dependencias, ejecute los siguientes comandos en la carpeta del taller:

```
python3.9 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Alternativamente, si el comando `python3.9 -m venv venv` no funciona, puede utilizar el comando `virtualenv -p python3.9 venv`.

Puede ejecutar el test suite del proyecto ejecutando el comando `./run_tests.sh`. Se recomienda la utilización de dicho comando para correr los tests ya que configura el entorno de `Python` para que los tests sean menos aleatorios. Para correr los casos de test en un archivo en particular, se puede pasar como primer argumento el módulo correspondiente. Por ejemplo, para ejecutar los tests en el archivo `test_ejercicio_5.py`, se puede ejecutar el comando `./run_tests.sh test.test_ejercicio_5`. Igualmente, también se pueden correr los tests en una IDE, como *PyCharm*, configurando la variable de entorno `PYTHONHASHSEED=0`.

Nota: en este taller no vamos a usar la librería de `coverage` de Python, ya que interfiere con la dependencia `fuzzingbook` utilizada en el taller.

Ejercicio 1

Implementar las siguientes funciones en el archivo `mutation_utils.py`:

```
def insert_random_character(s: str) -> str:
    """Retorna s con un caracter random insertado en una posicion al azar."""
    pass

def delete_random_character(s: str) -> str:
    """Retorna s con un caracter random eliminado."""
    """Si la cadena esta vacia, no la modifica."""
    pass

 def change_random_character(s: str) -> str:
    """Retorna s con un caracter modificado en una posicion al azar."""
    """El caracter a modificar es reemplazado por otro caracter random."""
    """Si la cadena esta vacia, no la modifica."""
    pass
```

Al elegir nuevos caracteres random, utilice la constante de Python `string.printable` que devuelve todos los caracteres ASCII "imprimibles" (dígitos, letras, símbolos y espacio en blanco).

Escriba tests para estas funciones en el archivo `test_mutation_utils.py`. Dado que las funciones a testear tienen un compontente aleatorio, utilice el método de Python `random.seed(semilla)` para configurar una semilla y hacer que los tests sean determinísticos.

Ejercicio 2

La clase `FunctionCoverageRunner` del paquete `fuzzingbook.MutationFuzzer` permite ejecutar un programa y luego obtener un conjunto con las líneas cubiertas durante su ejecución. Por ejemplo, sea el siguiente programa:

```
1: def crashme(s: str) -> None:
2:    if len(s) > 0 and s[0] == 'b':
3:       if len(s) > 1 and s[1] == 'a':
4          if len(s) > 2 and s[2] == 'd':
5:             if len(s) > 3 and s[3] == '!':
6:                raise Exception()
```

Si ejecutamos el siguiente código

```
input = "good"
crashme_runner = FunctionCoverageRunner(crashme)
result, outcome = crashme_runner.run(input)
locations = crashme_runner.coverage()
```

La variable `result` contendrá el resultado de ejecutar la función `crashme` con la entrada `good`, mientras que la variable `outcome` contendrá los valores `Runner.FAIL`, `Runner.PASS` o `Runner.UNRESOLVED` de acuerdo a como finalizó la ejecución del programa. Por otra parte, `locations` contendrá un conjunto de tuplas representando los pares ⟨nombreFuncion, numeroDeLinea⟩ cubiertos durante la ejecución del programa. Pueden encontrar este ejemplo y correrlo en el archivo `test_crashme.py`.

En este ejercicio se pide completar las funciones `get_contributing_inputs` y `get_covered_locations` de la clase `MagicFuzzer` en el archivo `magic_fuzzer.py` para que, dado un conjunto inicial de inputs y una función de entrada, ejecute estos inputs almacenando la cobertura total alcanzada por su ejecución.

```
class MagicFuzzer:
    def __init__(self, initial_inputs, function_to_call,
                 function_name_to_call = None) -> None:
        """Ejecuta inputs iniciales, almacenando la cobertura obtenida"""
        pass

    def get_contributing_inputs(self) -> List[str]:
       """Retorna la lista de los inputs que aumentaron la cobertura
       en el orden que fueron ejecutados"""
       pass

    def get_covered_locations(self) -> Set[Location]:
        """Retorna el conjunto de locaciones cubiertas de todas las ejecuciones"""
        pass
```

Completar la implementación de modo que se respete el siguiente comportamiento al correrlo para la función *crashme*. Puede correr los tests en el archivo `test_magic_fuzzer.py` para verificar que su implementación es correcta. Agregue tests si lo considera necesario.

initial_inputs	covered_locations	contributing_inputs
[]	{}	[]
[”*good*”]	{(”*crashme*”, 2)}	[”*good*”]
[”*bad*!”]	{(”*crashme*”, 2), (”*crashme*”, 3), (”*crashme*”, 4), (”*crashme*”, 5), (”*crashme*”, 6)}	[”*bad*!”]
[”*good*”, ”*goo*”]	{(”*crashme*”, 2)}	[”*good*”]
[”*good*”, ”*bad*!”]	{(”*crashme*”, 2), (”*crashme*”, 3), (”*crashme*”, 4), (”*crashme*”, 5), (”*crashme*”, 6)}	[”*good*”, ”*bad*!”]
[”*bad*!”, ”*good*”]	{(”*crashme*”, 2), (”*crashme*”, 3), (”*crashme*”, 4), (”*crashme*”, 5), (”*crashme*”, 6)}	[”*bad*!”]
[”*good*”, ”*b*”, ”*ba*”]	{(”*crashme*”, 2), (”*crashme*”, 3), (”*crashme*”, 4)}	[”*good*”, ”*b*”, ”*ba*”]
[”*good*”, ”*b*”, ”*bad*!”]	{(”*crashme*”, 2), (”*crashme*”, 3), (”*crashme*”, 4), (”*crashme*”, 5), (”*crashme*”, 6)}	[”*good*”, ”*b*”, ”*bad*!”]
[”*good*”, ”*go*”, ”*b*”, ”*bad*!”]	{(”*crashme*”, 2), (”*crashme*”, 3), (”*crashme*”, 4), (”*crashme*”, 5), (”*crashme*”, 6)}	[”*good*”, ”*b*”, ”*bad*!”]

Ejercicio 3

La función `getPathID` del paquete `fuzzingbook.GreyboxFuzzer` retorna un valor de hash de 128 bits dado un conjunto de locaciones cubiertas. Vamos a utilizar ese valor como identificador de un camino en un programa (aunque dado que FunctionCoverageRunner.coverage() retorna un conjunto no es posible obtener el orden en que se recorre la secuencia de locaciones).

Sea $e(s)$ la siguiente función

$$e(s) = \frac{1}{\#Aparaciones(getPathID(s))^a}$$

donde a es un exponente positivo, y $\#Aparaciones$ es la cantidad de veces que se observó el $pathID$ durante la ejecución del fuzzer.

Escribir la implementación de la siguiente clase que almacena la *energía* de cada input ejecutado y permite elegir un input de acuerdo a cuanta energía tiene asignada. Esta clase se encuentra en el archivo `roulette_input_selector.py`.

```
class RouletteInputSelector:
    def __init__(self, exponent: int):
        """Guarda el exponente"""
        pass

    def add_new_execution(self, s: str, s_path: Set[Location]):
       """Agrega una nueva ejecucion del input s y su path"""
       pass

    def get_frequency(self, s:str) -> int:
       """Retorna la cantidad de apariciones del path_id de s"""
       """Retorna 0 si el input no fue ejecutado"""
       pass

    def get_energy(self, s:str) -> float:
       """Retorna el valor actual de e(s)"""
       """Levanta una excepcion si el input no fue ejecutado"""
       pass

    def select(self) -> str:
       """Elije aleatoriamente un s usando seleccion de ruleta sobre e(s)"""
       pass
```

Por ejemplo, luego de ejecutar la función `add_new_execution` con los siguientes parámetros

1. $s =$ ”*good*”, s_path=[[”*crashme*”, 2]]

2. $s =$ "goo", s_path=[["$crashme$", 2]]
3. $s =$ "go", s_path=[["$crashme$", 2]]
4. $s =$ "b", s_path=[["$crashme$", 2], ["$crashme$", 3]]
5. $s =$ "bx", s_path=[["$crashme$", 2], ["$crashme$", 3]]
6. $s =$ "by", s_path=[["$crashme$", 2], ["$crashme$", 3]]
7. $s =$ "bad", s_path=[["$crashme$", 2], ["$crashme$", 3], ["$crashme$", 4], ["$crashme$", 5]]

La función get_frequency debe contener los siguientes valores

s	get_frequency
"$good$"	3
"goo"	3
"go"	3
"b"	3
"bx"	3
"by"	3
"bad"	1

Mientras que, con un exponente de 2, la energía de cada input es:

s	get_energy
"$good$"	$\frac{1}{3^2}$
"goo"	$\frac{1}{3^2}$
"go"	$\frac{1}{3^2}$
"b"	$\frac{1}{3^2}$
"bx"	$\frac{1}{3^2}$
"by"	$\frac{1}{3^2}$
"bad"	$\frac{1}{1^2}$

Entonces, la probabilidad de elegir a "$good$","goo","go","b","bx" o "by" es

$$\frac{\frac{1}{3^2}}{6 \times (\frac{1}{3^2}) + \frac{1}{1^2}}$$

Mientras que la probabilidad de elegir a "bad" es:

$$\frac{\frac{1}{1^2}}{6 \times (\frac{1}{3^2}) + \frac{1}{1^2}}$$

Puede correr los tests en el archivo `test_roulette_input_selector.py` para verificar que su implementación es correcta. Agregue tests si lo considera necesario.

Ejercicio 4

Implementar las funciones `fuzz` y `mutate` de la clase `MagicFuzzer` para que realice una selección de ruleta de un nuevo input para mutar y luego ejecutar, actualizando debidamente todas sus estructuras internas. Además, implementar la función `run_until_covered` que corre una campaña de fuzzing hasta alcanzar el cubrimiento de todas las líneas del programa objetivo. Inspeccione el archivo `get_source_lines.py` para ver como obtener el total de líneas de un programa.

```
class MagicFuzzer:
   def mutate(self, s: str) -> str:
      """Aplica al azar alguna de las tres operaciones de mutacion definidas antes"""
      pass

    def fuzz(self):
       """
       Elije aleatoriamente un input s usando seleccion de ruleta sobre e(s),
       muta el input s utilizando la funcion mutate(s), y ejecuta el s mutado
       """
       pass

   def run_until_covered(self) -> int:
      """
      Corre una campania del MagicFuzzer hasta cubrir todas las lineas del programa.
      Retorna la cantidad de iteraciones realizadas.
      """
      pass
```

Ejecutar 3 campañas de fuzzing hasta alcanzar el cubrimiento de todas las líneas del programa `crashme` a partir de input inicial "" (i.e. string vacío). Utilice exponente 2 para la selección de ruleta. Modifique el archivo `test_ejercicio_4.py` para dejar constancia de los resultados obtenidos. El test debe indicar la cantidad de inputs necesarios para cubrir todas las líneas del programa en cada campaña. Utilice una semilla (distinta para cada campaña) en los tests para que los resultados sean determinísticos.

Nota: si ve que la campaña tarda mucho en converger, pruebe utilizando otra semilla.

Ejercicio 5

Modificar el `MagicFuzzer` para que, si no recibe el parámetro `function_name_to_call`, no filtre las locaciones de acuerdo a la función que se desea cubrir. Además, implementar la función `run_iterations(n)` que corre una campaña de fuzzing por n iteraciones.

```
class MagicFuzzer:
   def run_iterations(self, n: int) -> None:
      """
      Corre una campania del MagicFuzzer por n iteraciones.
      """
      pass
```

Ejecutar 3 campañas de fuzzing usando $n = 5000$ sobre la función `my_parser` del archivo `my_parser.py` usando como input inicial "" (i.e. string vacío), `function_name_to_call` el parámetro vacío. Utilice exponente 2 para la selección de ruleta. Modifique el archivo `test_ejercicio_5.py` para dejar constancia de los resultados obtenidos. El test debe indicar la cantidad máxima de líneas cubiertas del programa en cada campaña. Utilice una semilla (distinta para cada campaña) en los tests para que los resultados sean determinísticos.

Nota: en algunas ocasiones se ha observado que la función `my_parser` puede llegar a comportarse de manera no determinística, incluso utilizando una semilla para el random. Si ese fuera el caso, utilice el script `run_tests.sh` para correr los tests de este ejercicio o configure la variable de entorno `PYTHONHASHSEED=0`.. Además considere implementar cada campaña de fuzzing en un test distinto para evitar que la ejecución de una campaña afecte a la siguiente.

Formato de Entrega

El taller debe ser entregado en el campus de la materia. La entrega debe incluir un archivo `entrega.zip` con el código implementado. Este debe estar detalladamente documentado, incluyendo en la documentación una descripción de la resolución de cada ejercicio, con una breve discusión de las decisiones de diseño más importantes tomadas para resolver el taller.