Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC3745: Testing
Tarea 1: Lexical Fuzzing
Juan Pablo Sandoval & Maximiliano Narea

Martes 13-Agosto-2024

1 Objetivos

El fuzzing nació en una "oscura y tormentosa noche de verano en 1988" [Takanen et al., 2008]. Sentado en su apartamento en Madison, Wisconsin, el profesor Barton Miller estaba conectado a su computadora en la universidad a través de una línea telefónica de 1200 baudios. La tormenta provocó interferencias en la línea, lo que causó que los comandos UNIX recibieran entradas incorrectas, llevando a que los programas se cerraran abruptamente (crash). La frecuencia de estos fallos sorprendió a Miller: **¿no deberían los programas ser más robustos?** Como científico, decidió investigar la gravedad del problema y sus causas. Así que diseñó una tarea de programación para sus estudiantes en la universidad, en la que desarrollarían los primeros fuzzers [fuzzing book[1]].

En el curso, se abordaron estrategias sencillas para medir la cobertura, generar entradas aleatorias, crear entradas mediante mutadores, y emplear algoritmos de búsqueda para encontrar entradas que cumplan ciertas condiciones. El objetivo de esta tarea es poner a prueba el conocimiento adquirido en clase. La tarea se divide en tres partes, cada una evaluando de manera independiente un tema visto en clase.

- **Parte A: Cobertura de Ramas (Branch Coverage):** El objetivo de la primera parte es que los y las estudiantes comprendan el código visto en clase y lo adapten para identificar automáticamente las ramas cubiertas por las pruebas.
- **Parte B: Fuzzing Basado en Aleatoriedad y Mutación:** El objetivo es que los y las estudiantes apliquen técnicas de prueba aleatoria y prueba basada en mutación para analizar una función y encontrar entradas que maximicen la cobertura de declaraciones (statement coverage).
- **Parte C: Fuzzing Basado en Búsqueda:** El objetivo es que los y las estudiantes desarrollen un algoritmo de búsqueda que permita encontrar los valores necesarios para cumplir con las condiciones dentro de una función, comenzando con una entrada aleatoria y aproximándose a la solución mediante una función objetivo.

[1]Fragmento de Texto Tomado del Libro "The Fuzzing Book", Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler

2 Descripción

2.1 Parte A: Cobertura de Ramas (Branch Coverage)

En clase, creamos una clase llamada `Coverage` que contenía dos métodos útiles:

- `trace`: Este método devolvía una lista con las líneas ejecutadas en el orden de ejecución[2]
- `coverage`: Este método devolvía una lista con las líneas ejecutadas al menos una vez, eliminando duplicados. Incluso si una línea se ejecutaba varias veces, solo aparecía una vez en la lista.

Considere la siguiente función ejemplo:

```
1  def categorize_product_quality(rating):
2      # Ensure the rating is within the valid range
3      if rating < 1 or rating > 100:
4          return "Invalid rating"
5      # Categorize based on rating
6      if rating <= 20:
7          return "Poor"
8      elif rating <= 40:
9          return "Fair"
10     elif rating <= 60:
11         return "Good"
12     elif rating <= 80:
13         return "Very Good"
14     else:
15         return "Excellent"
```

Por ejemplo, podemos utilizar la clase `Coverage` para verificar qué líneas de código cubren los siguientes dos tests, como se ve a continuación:

```
import fuzzingbook.bookutils.setup
from fuzzingbook.Coverage import Coverage
with Coverage() as cov:
    assert categorize_product_quality(50) == "Good" # test 1
    assert categorize_product_quality(20) == "Poor" # test 2
# la función coverage devuelve las lineas ejecutadas sin duplicados, pero en desorden, la ordenamos
sorted_executed_lines = sorted(cov.coverage(), key=lambda x: (x[1]))
# filtramos las lineas ejectudas en categorize_product_quality
covered_lines = list(filter(lambda item: item[0] == 'categorize_product_quality', sorted_executed_lines))
# imprimimos
for item in covered_lines:
    print(item)
```

El ejemplo anterior imprime el siguiente resultado:

```
('categorize_product_quality', 3)
('categorize_product_quality', 6)
('categorize_product_quality', 7)
('categorize_product_quality', 8)
('categorize_product_quality', 10)
('categorize_product_quality', 11)
```

Como se vio en clase, la función `trace` devuelve más datos "de lo esperado", ya que Python no solo ejecutó el método `categorize_product_quality`, sino también otros métodos. Para mostrar solo las lineas cubiertas del método bajo analisis, aplicamos un filtro que considera únicamente las líneas del método `categorize_product_quality`.

Problema: El problema con el line coverage es que, en ocasiones, se puede obtener un 100% de cobertura sin haber ejecutado todos los posibles flujos de ejecución. Por ejemplo, considere la siguiente función:

```
def example(input):
    if input > 10:
        print("yes")
```

[2]Si una línea se ejecutaba más de una vez, aparecía múltiples veces en la lista.

Si realizamos un test con un input mayor a 10, obtendremos un 100% de cobertura, pero sin evaluar el caso en que el input sea menor o igual a 10, lo que genera una falsa sensación de completitud.

Tarea: Cree una clase `BranchCoverage` que herede de la clase `Coverage` y sobrescriba el método `coverage`. Este método debe devolver cada par de líneas que se ejecutaron una seguida de la otra (secuencialmente) al menos una vez, eliminando duplicados. Por ejemplo:

```
with BranchCoverage() as bcov:
    assert categorize_product_quality(50) == "Good"  # test 1
    assert categorize_product_quality(20) == "Poor"  # test 2
covered_pairs = bcov.coverage()
covered_pairs = list(filter(lambda item: item[0][0] == 'categorize_product_quality', covered_pairs))
for item in covered_pairs:
    print(item)
```

La salida debería ser la siguiente:

```
(('categorize_product_quality', 3), ('categorize_product_quality', 6))
(('categorize_product_quality', 6), ('categorize_product_quality', 7))
(('categorize_product_quality', 6), ('categorize_product_quality', 8))
(('categorize_product_quality', 7), ('_internal_set_trace', 48))
(('categorize_product_quality', 8), ('categorize_product_quality', 10))
(('categorize_product_quality', 10), ('categorize_product_quality', 11))
(('categorize_product_quality', 11), ('categorize_product_quality', 3))
```

Observe que ahora la lista no solo devuelve las lineas que fueron ejecutadas al menos una vez, sino que una línea puede aperecer dos veces si la misma alteró el flujo de ejecución. Por ejemplo, la línea 6, que es un `if`, aparece en la lista dos veces la primera cuando ingreso al cuerpo del `if`, y la segunda cuando no.

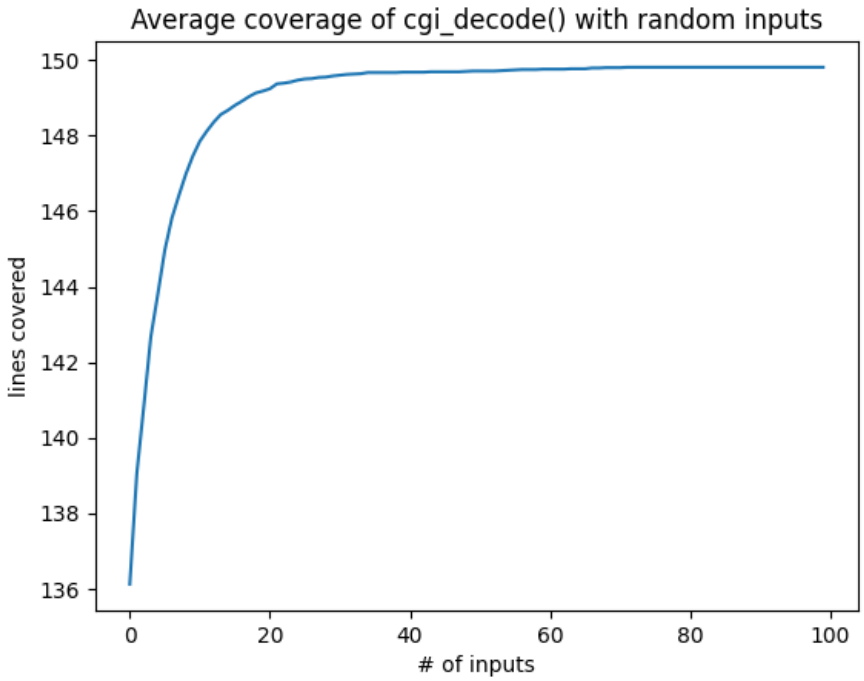
Adjunto a este enunciado podrá encontrar un *Jupyter Book*, que tiene una celda vacía, donde usted puede escribir su solución. Además, al final, el *Jupyter Book* tiene código que permitirá probar su solución.

2.2 Parte B: Fuzzing Basado en Aleatoriedad y Mutación

En este ejercicio, usted tendrá que utilizar la técnica de *Mutation Based Fuzzing* para generar entradas para la siguiente función, con el objetivo de obtener entradas que logren ejecutar la mayor cantidad de líneas al menos una vez (coverage):

```
def validate_device_config(config: bitarray) -> bool:
    # Check length
    if len(config) != 12:
        return False
    mode = config[:2].to01()
    if mode == '00':
        # Mode 00: Configuration is invalid if next 2 bits are not '11'
        if config[2:4].to01() != '11':
            return False
    ...
    return True
```

En el *Jupyter Book* que acompaña el enunciado, podrá encontrar un script que permite generar entradas aleatorias para esta función. La gráfica a continuación muestra el coverage obtenido después de utilizar 100 entradas generadas aleatoriamente. Recuerde que, para generar esta gráfica, acumulamos el coverage a medida que ejecutamos cada entrada aleatoria. Es decir, al ejecutar las 100 entradas aleatorias, logramos obtener 150 líneas ejecutadas. Las últimas 70 entradas aleatorias no lograron ejecutar ninguna línea nueva, lo que explica la forma de la curva.



Tarea: Su tarea consiste en utilizar *mutation based fuzzing* para generar entradas a partir de una entrada semilla. En particular, debe crear mutadores que, dado un `bitarray`, devuelvan un nuevo `bitarray` mutado. Usted tiene la libertad de diseñar los mutadores que desee; por ejemplo, cambiar un bit de 1 a 0. Debe crear al menos 6 mutadores.

En el *Jupyter Notebook* ya existen algunos scripts que le permitirán probar la efectividad de su solución. Estos scripts toman como entrada los mutadores que usted cree y los utilizan para mutar una entrada semilla, con el objetivo de obtener valores de entrada que mejoren el coverage. Una vez que implemente sus mutadores y observe los resultados, deberá discutir si su solución es más efectiva que generar entradas aleatorias y también cual de sus mutadores tiene un mejor desempeño y por qué. Esta discusión debe ir al final el *Jupyter Book*.

2.3 Parte C: Fuzzing Basado en Búsqueda

En esta parte de la tarea, usted debe adaptar el algoritmo `hillclimb`, visto en clase, para encontrar tres valores que logren pasar la validación de la siguiente función:

```
def safety_check(temp, pressure, humidity):
    if temp >= 100:
        return False
    if pressure >= 200:
        return False
    if humidity < 20 or humidity > 70 :
        return False
    if temp > 90 or pressure >= 180:
        return False
    return True
```

Recuerde que este algoritmo comienza probando tres datos de entrada aleatorios y luego ajusta los valores para encontrar aquellos que hagan que la función, en este ejemplo, devuelva `True`. Para ayudarlo en su tarea, hemos adjuntado un *Jupyter Book* que incluye una plantilla que le será útil para desarrollar su solución. La plantilla tiene pasos faltantes que usted debe completar:

A) Defina la función `neighbors`: Esta función recibe tres valores de entrada y devuelve una lista de vectores de tres valores (los vecinos). Por ejemplo:

```
print(neighbors(10, 10, 10))
```

Debe generar un vector similar a este:

```
[(9, 9, 9), (9, 9, 10), (9, 9, 11), ...]
```

Usted puede implementar la estrategia que prefiera para obtener los vecinos, por ejemplo, sumando y restando uno a los valores recibidos de entrada. Los valores de los vecinos no deben ser mayores a 1000 ni negativos, es decir, deben variar entre 0 y 1000.

B) Defina una Función Fitness: En este ejercicio utilizaremos el *branch-distance* como función *fitness*. Recuerde que el *branch-distance* es un valor que indica qué tan cerca están los valores de entrada de cumplir una condición. Por ejemplo, considere la condición $a > 100$ con $a = 20$. Esta condición debería devolver `False`, y la distancia para que devuelva `True` es 81, ya que si sumamos 81 a 20, obtendremos 101, y la condición sería `True`. Por otro lado, la distancia es 0 cuando la condición ya evalúa como `True`.

Cada vez que ejecutamos la función a evaluar, debemos calcular el *fitness*. Con este objetivo, le ofrecemos la versión instrumentada del método `safety_check`. La idea de tener una versión instrumentada del método es poder calcular el *fitness* sin alterar su funcionamiento original.

```
def instrumented_safety_check(temp, pressure, humidity):
    if evaluate_condition(1, '>=', temp, 100):
        return False
    if evaluate_condition(2, '>=', pressure, 200):
        return False
    if (evaluate_condition(3, '<', humidity, 20) or evaluate_condition(4, '>', humidity, 70)):
        return False
    if evaluate_condition(5, '>', temp, 90) or evaluate_condition(6, '>=', pressure, 180):
        return False
    return True
```

En esta tarea, usted debe implementar primero la función `evaluate_condition`. Esta función, cada vez que se evalúe una condición, debe devolver el resultado de evaluar dicha condición (`True` o `False`), de manera que la función `instrumented_safety_check` funcione igual que la original. Además de devolver el valor, la función `evaluate_condition` debe guardar el *branch distance* de cada condición en una variable global (por ejemplo, un diccionario). Note que la distancia que nos interesa en este ejercicio es la distancia para condiciones que evalúen como `False`.

```
def evaluate_condition(id, op, a, b) -> Bool:
    ...
```

Como segundo paso, se debe implementar la función `get_fitness_validation`, que, dados los tres argumentos de la función `instrumented_safety_check`, devuelva el *branch distance* total, que es la suma de las *branch distances* de todas las condiciones:

```
def get_fitness_validation(x,y,z):
    try:
        # Run the function under test
        # It saves the branch distances on a global variable
        instrumented_safety_check(x,y,z)
    except BaseException:
        pass
    # read the data saved in the global variable and sum up branch distances
    # save this value in a fitness variable
    ...
    return fitness
```

Para saber si fitness función esta bien implementada, esta debe devolver 0 si los tres numeros ingresados como argumentos pasan todas las condiciones y la función `safety_check` devuelve `True`. Si los valores esta cerca de cumplir las condiciones deben devolver un valor cercano a cero, si se aleja debe devolver un valor mas grande. Puede probar esto, ejecutando la función con varios valores y ver si funciona. Por ejemplo:

```
get_fitness_validation(0,0,70)
get_fitness_validation(20,20,79)
```

Al final del *Jupyter Book*, encontrará un script que permite ejecutar el algoritmo de búsqueda. Si su solución es correcta, el algoritmo, dado tres valores aleatorios, debería poder iterar y llegar a un conjunto de tres valores que cumplan las condiciones. En cada iteración la función fitness deberia ir reduciendo hasta llegar a 0. También hay un script que ejecuta el algoritmo 100 veces, devolviendo el promedio de iteraciones que el algoritmo necesita para encontrar la solución.

Recomendaciones:

- Se recomienda normalizar las *branch distances* antes de sumarlas, por ejemplo, dividiéndolas entre 1000 (siendo 1000 el valor máximo para cada valor).
- Cada persona puede implementar de manera diferente el cálculo del *fitness*, siempre y cuando sea útil para el algoritmo y lo ayude a encontrar la solución.
- No se permiten funciones de *fitness* hardcodeadas o sesgadas a la función `safety_check`. Revisaremos su tarea utilizando diferentes valores en las condiciones, por ejemplo, cambiando el primer `if` de $temp \geq 100$ a $temp \geq 50$.

3 Código Base

GitHub Classroom. Para realizar la actividad, debe unirse al repositorio de GitHub Classroom su sección del curso y aceptar la tarea nř1.

- *Sección 1:* `https://classroom.github.com/a/bKAKlkYP`
- *Sección 2:* `https://classroom.github.com/a/WCV7bIMH`

Lo anterior le creará un repositorio con la siguiente estructura de carpetas para la actividad:

```
T1_Coverage.ipynb
T1_MutantBase.ipynb
T1_SearchBase.ipynb
```

Evaluación. Su tarea será revisada manualmente por el cuerpo de ayudantes. Es por esto que se sugiere comentar su código y explicarlo de manera clara cuando corresponda.

4 Entregable, Fecha de Entrega y Penalizaciones

Deberá renombrar los diferentes jupyter notebooks con su número de alumno, manteniendo el identificador de cada jupyter, por ejemplo para el primer archivo `T1_Coverage.ipynb`, el formato a entregar sería `17626465_Coverage.ipynb`. Debe subir el código los archivos modificados al repositorio asignado para su tarea.

- **Fecha de Entrega:** Viernes 6 de Septiembre hasta las 23:59.
- **Aclaración** La tarea esta pensada para que se resuelva en dos semanas, se publica el enunciado con tres semanas anticipación para los que quieren ir avanzando y organizandose con sus otras materias.
- **Retrasos** Recuerde que cada persona tiene una tarjeta que le habilita a tener tiempo extra para una de las cuatro tareas. La tarjeta solo puede utilizarse una vez.
- **Premio** Este semestre premiaremos las 3 mejores tareas, para rankear las tareas, consideraremos los mutadores y funciones fitenss mas originales y efectivos. Primer lugar se ganara 3 decimas extras al examen, segundo lugar 2 decimas, y tercer lugar 1 decima. Ademas el primer lugar tendra un premio comprado de almacen UC.