A tool for analysing Python programs based on Chef

Supervisor: A.Prof. Truong Anh Hoang
Student: Nguyen Thanh Toan

University of Engineering and Technology, VNU

May 21, 2015

Table of Contents

Background

- Unit testing
- Symbolic execution
- Concolic testing

Unit testing

- Unit testing is a method that we check an unit or module of a program.
- Users create input values, run them on the unit and then compare result with their expected outcomes.

Unit testing

- Unit testing benefits to find problems early, to facilitate changes, to simplify integration testing and to provide documentation, software design.

```
import unittest
def average(x, y):
    return (x + y)/2

class AverageTest(unittest.TestCase):
   def test_1(self):
      result = average(3, 5)
      expected_result = 4
      self.assertEqual(result, expected_result)

   def test_2(self):
      result = average(3, 4)
      expected_result = 3
      self.assertEqual(result, expected_result)
```

Symbolic execution

- Instead of using concrete values, symbolic execution utilizes symbols to cover more paths in program.
- Existing symbolic execution engines: KLEE on LLVM, JPF on Java, Jalangi on JavaScript.

```
x = input("Enter a number")
if x > 3:
    print "x is greater than 3"
else:
    print "x is equal or less than 3"
```

Concolic testing

- Combining concrete testing and symbolic execution
- Utilizing advantages and minimizing disadvantages of these two techniques

```
def function(x, y):
    z = 2*y
    if x == 10000 :
        if x < z:
            assert(0) #error
```

S2E framework

- One problem of symbolic execution is that how programs interact with their environment
- S2E creates a virtual machine and performs symbolic execution inside it.
- S2E has been used for:
  - Automated testing
  - Reverse engineering
  - Performance profiling

Chef recipe

- Chef proposes a recipe to adapt interpreted programs to run on S2E framework.
- The problem between interpreted and low-level language is statement coverage.
- The solution of Chef is Class Uniform Path Analysis

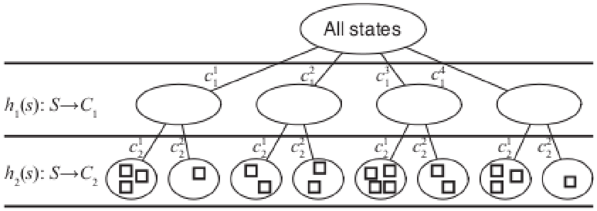

Figure: CUPA state partitioning

Installing Chef tool

- Chef installation involves three different documents that are not unified.
- We combines them into one unified installation guide:
  - Installing S2E framework
  - Creating Chef virtual machine
  - Setting up host and guest repositories
  - Running symbolic execution on Python programs
- Users can follow this document guide to install and run Chef straightforward.

Running Chef results

- We analyse the number of high-level and low-level test cases and it is possible to complete after 6 hours running
- Chef engine power: 512 GB RAM
- Our engine power: 8GB RAM

Test	HL test cases	LL test cases	Completed
ArgparseTest	64	303	yes
ConfigParserTest	65	4543	no
HTMLParserTest	476	5238	no
SimpleJSONTest	21	1907	no
XLRDTest	2492	2730	no
UnicodeCSVTest	164	208314	no

Table: Testing result of 6 Python tests

Analysing Chef results

- Chef advantages:
  - Chef is capable of running symbolic execution directly on interpreted programs such as Python, Lua.
  - Chef can build symbolic execution engine for Python in 8 days and Lua in 5 days.
  - Chef symbolic execution engines are not weaker than manual built ones.
- Chef limitations:
  - Chef only experiments symbolic execution on **getString** function.
  - Chef is performed on powerful machine with 512 GB while it usually gets stopped on personal computers.
  - To build Chef symbolic execution engine, it needs to understand S2E framework thoroughly.

Tool overview

- This tool generates large quantity of input values that are hard for individuals to create themselves.
- It also completes about 80 per cent work of writing unit test cases for developers.
- It utilizes the result of running 6 programs on Chef symbolic execution engine.

Generating test case procedure

- Modifying the format of input programs
- Eliminating invalid input values
- Generating unit test cases

Modifying input programs

- The input programs are created to instrument to Chef symbolic execution engine.

```
class HTMLParserTest(light.SymbolicTest):
    def setUp(self):
        self.HTMLParser =
            importlib.import_module("HTMLParser")

    def runTest(self):
        parser = self.HTMLParser.HTMLParser()
        parser.feed(self.getString("html", '\x00'*15))
        parser.close()
```

Modifying input programs

- They need to be transformed so that they can take concrete values to run.

```
import HTMLParser

class HTMLParserTestFunction(unittest.TestCase):
    parser = HTMLParser.HTMLParser()
    parser.feed(input_string)
    parser.close()
```

Eliminating invalid input values

- Input values that are not complete.

```
2740316685 0xb760b396 arg2_name.s#value=>"---"
    arg1_name.s#value=>"-\x00-"
834885621 0xb760b396 arg1_name.s#value=>"---"
```

- Input values that are the same, especially the null string.

Generating unit test cases

```
import unittest
import unicodecsv
import cStringIO

class UnicodeCSVTestFunction(input_string):
      f = cStringIO.StringIO(input_string)
      r = self.unicodecsv.reader(f, encoding="utf-8")
      for row in r
         pass
      f.close

class HTMLParserTest(unittest.TestCase):
   def test_1(self):
      result = UnicodeCSVTestFunction(",,\n,,")
      self.assertEqual(result, expected_result)
```

Tool evaluation

- A large amount of input values are generated for unit testing.
- Our tool can generate hundreds to thousands test cases
- We complete about 80 per cent of writing unit test work

Test	Generated test cases
ArgparseTest	293
ConfigParserTest	4540
HTMLParserTest	5236
SimpleJSONTest	1905
XLRDTest	2720
UnicodeCSVTest	208214

Table: Testing result of generating unit test cases

Conclusions

- We create a tool that generate a large number of input values for Python programs
- We also analyse the Chef tool and it is possible to apply to other interpreted languages
- We unify three different installation guide into one Chef installation and running document

Future Work

- Automatically computing value of expected_result variable
  - Test cases can be runnable
- Applying Chef recipe to JavaScript
  - Mastering S2E plug-in construction
  - Setting up on of ECMASCript Engines as interpreter
  - Comparing with existing work of Kudzu and Jalangi