Department of Computer Science
School of Science and Engineering, LUMS

Senior Year Project Report

PyFix

Fault Localization & Automated Program Repair in *Python*

Muhammad Zain Qasmi - 18100276

Muhammad Dawood Jehangir - 18100164

Ali Ahsan - 18100071

June, 2017 - May, 2018

A report submitted in part fulfilment of the degree of

BS in Computer Science

Supervisor: Junaid Haroon Siddiqui

Problem Statement

Despite of so much resources being spent on developing and testing programs, programs today with their ever increasing scale and complexity tend to contain bugs which are difficult to track and fix - making it impractical for programmers to manually locate them. Hence, our project focuses on making debugging for programmers easier by localizing the bug and synthesizing program repairs.

Software maintenance, of which bug repair is a major component, is time-consuming and expensive, accounting for as much as 90 percent of the cost of a software project at a total cost of up to $70 billion per year in the US. Put simply: Bugs are ubiquitous, and finding and repairing them are difficult, time-consuming, and manual processes.

Our tool aims to predict the most suspicious statements inside a Python program, modifies them based on the cues available from the rest of the program and verifies if that fixes the program.

Table of Content

Problem Statement **2**

Introduction **3**

Source Code **4**

Description **4**

Fault Localization 4

Importance of an effective fault localization tool 4

Why we favored Tarantula 5

Program Repair 5

Why is Automated Program Repair challenging 5

Project Specification **5**

The Structure of PyFix Pipeline 6

The Algorithm 6

Fault Localization 6

Implementation Details of TarantulaPy 6

Program Repair 8

Dividing script into distinct code blocks 8

Computing Similarity of code blocks with bugged block 8

Repairing Code 8

i) Permuting and mapping variables
ii) Permuting arithmetic operators
iii) Permuting bool statements 8

Testing new script 9

Getting Started **9**

Checking the results 10

Future Work/Limitations **11**

Other Work **12**

Mimicking KLEE Symbolic Execution Engine in Python 12

Source Code 12

What's KLEE? 12

Our Work in Python 12

Why we had to forgo this Project 12

References

13

Introduction

Techniques for automatically detecting software flaws include intrusion detection, model checking and lightweight static analyses and software diversity methods. However, detecting a defect is only half of the story: Once identified, a bug must still be repaired. As the scale of software deployments and the frequency of defect reports increase, some portion of the repair problem must be addressed automatically.

For this purpose we propose PyFix - a technique that uses existing test cases to automatically generate repairs for real-world bugs in simple Python programs. We define a repair as a patch consisting of one code change that, when applied to a program, causes it to pass a set of test cases which typically includes both tests of required behavior as well as a test case encoding the bug. The test cases may be human written, taken from a regression test suite, steps to reproduce an error, or generated automatically.

Source Code

https://github.com/ZainQasmi/Automated-Program-Repair

Description

The problem of automatically fixing programming errors is a very active research topic in software engineering. This is a challenging problem as fixing even a single error may require analysis of the entire program. In practice, a number of errors arise due to programmer's inexperience with the programming language or lack of attention to detail. We call these common programming errors. These are analogous to grammatical errors in natural languages.

Compilers detect such errors, but their error messages are usually inaccurate. In this work, we present an end-to-end solution, called PyFix, that can fix many such errors in a program without relying on any external tool to locate or fix them. At the heart of PyFix is a dual-layered engine the first (TarantulaPy) part of which predicts erroneous statements in bugged programs while the second (Program Repair) part attempts to remedy them.

Fault Localization

Importance of an effective fault localization tool

A fault localization technique directs a programmer's attention to specific parts of a program. Given one or more failing test cases and zero or more passing test cases, a FL technique outputs a (typically, sorted) list of suspicious program locations, such as lines, statements, or declarations. The FL technique uses heuristics to determine which program locations are most suspicious—that is, most likely to be erroneous and associated with the fault. A programmer can save time during debugging by focusing attention on the most suspicious locations.

Traditional methods of fault localization relied either on analyzing memory dumps which inarguably requires processing a huge cache of data or on inserting print statements by making intelligent guesses to center on the bug. However, this requires the user to have a good understanding of the program and is not feasible for large-scale programs even with the use of tools such as Microsoft Visual Studio debugger. Since the primary problem of isolating code based on its susceptibility to fault is still unresolved.

Most modern fault localization methods work by focusing on prioritizing suspicious code based on its likelihood of containing faults which the programmers manually checks on later whether it indeed contains bugs - since reducing this search space is critical to easily isolating bugs.

Why we favored Tarantula

The original Tarantula is a visual debugging tool for programs written in the C language and uses SFL. Each Line of Code (LOC) is highlighted accordingly the probability calculated when Tarantula runs the test suits against the System Under Test. This highlight is made using a colour gradient from green to red. The green means the minimum suspiciousness and the red means the maximum suspiciousness.

There were few other options available as well such as Nearest Neighbour, Cause Transitions and Set Union and Set Intersection. Set Union, Set Intersection and Nearest Neighbour were rejected since the paper on fault localization techniques[1] argued they are less effective because of their sensitivity to particular test suites. Cause transition was an intriguing choice since it performs a binary search between a passing test case and failing test case. It does perform better than the three mentioned before but empirical studies have shown that Tarantula outperforms Cause Transition consistently.

Based on the reasons provided above, we decided to implement Tarantula for Python (we name in TarantulaPy), which took the key position of fault localization tool in our PyFix pipeline.

Program Repair

This project works towards enhancing both effectiveness and efficiency in order for APR to be practically adopted in foreseeable future. However much needs to be done and in the future, we plan to synergize many existing APR techniques, improve our predictive model, and adopt the advances of other fields such as test case generation and program synthesis for APR. [5]

Why is Automated Program Repair challenging

APR has recently been an emerging research area, addressing an important challenge in software engineering. PyFix has a backbone based on search based APR techniques which if effective and efficient, can greatly help software debugging and maintenance. However, to produce repairs, search based APR techniques generate huge populations of possible repairs, i.e., search space, and lazily search for the best one among the search space. Despite recent advances, search based APR techniques still suffers from search space explosion problem. Furthermore, APR techniques may be subject to overfitting, in which generated repairs do not generalize to other test sets.

Project Specification

On a high level PyFix is decoupled into two distinct modules, the first of which - Tarantula - attempts to localize the bugged line and passes it on to the second Program Repair module which attempts to repair the bug and returns a bug-free code with suggested fixes that atleast passes all the given test cases.

PyFix takes as input the source code containing a defect (**buggedProg**), a set of test case results (**testResults**) which includes the input arguments to the program and whether the case passed or failed, and lastly a set of test cases (**testCases**) against which our newly generated program will be tested for success.

The Structure of PyFix Pipeline

1. TarantulaPy: Computes the coverage information against multiple test cases, evaluates the suspiciousness score of the lines of code and ranks them. Line with the highest rank is most likely to have a bug.
2. Program Repair: Iterates over the ranked statements provided by TarantulaPy. Separates script into distinct blocks and computes differential between buggy blocks and other blocks of code. Ones with highest similarity are selected to extract probable differences and incorporate them in code variants. Variants are then tested against all test cases. Iteration breaks when either a variant passes all test cases pass or when all suggested variants are exhausted.

The Algorithm

Input :

- A python program with bug(s)
- A file with test case results for Tarantula
- A suite of Test Cases to test against new code variants

Output :

- Python program with problem fixed OR
- Python program with bug(s) (Signifying our tool couldn't rectify the error)

Begin :

- Run TarantulaPy to get the list of most suspicious statements of code.
- Select top few lines for processing. (Depends on how many user sets).
- Select one line of code from the pool and extract the block around it. Block size is again determined by the user. (We call this block block of interest)
- The similarity of block of interest is compared with other block using Levenshtein distance. Few most similar blocks are picked. (In our case 3 is the block size)

- Different variants of the program are generated by modifying the identified buggy line. They're evaluated against all the test cases provided.
- Process ends when either all the test cases pass for a variant or all the potentially erroneous lines are exhausted.

Fault Localization

Implementation Details of TarantulaPy

We implemented the famous fault localization technique known as Tarantula such that an automated Python program can be used to identify bugs in a given python script. In our implementation we computed the suspiciousness score of a line of code's likelihood for bug using the formula given in the paper i.e.

$$suspiciousness(e) = \frac{\frac{failed(e)}{total\ failed}}{\frac{passed(e)}{total\ passed} + \frac{failed(e)}{total\ failed}}$$

The critical component of Tarantula is the mechanism/algorithm used to give a "likelihood" score to every statement in the code. One approach by Tarantula was to calculate the ratio of test cases passed and test cases failed by a statement and sort them by this score. However, this technique is limited by the number of test cases available - multiple failed test cases let the technique leverage the richer information base. Also, this likelihood approach ignores the program structure and dependencies between program entities rather focusing only on tracing execution for failed cases.

A dictionary datastructure was used to cater the ranks of each of line of code using "key" as line number and its "value" is the ranking based on suspiciousness score. The suspiciousness score for each line was calculated and sorted such that the line with highest score was given a rank of 1 and should be evaluated first and foremost for bug checking. If that statement is found to not be faulty, then the statement with the next rank is evaluated.

To test which lines were hit by the code, we used Python's settrace() method from sys library which takes three arguments - the stack frame; event; and an argument frame which is the current stack frame. An event simply returns a 'call' or 'line' string etcetera based on which the function decides what to do such as in case of a 'call' string the trace function is invoked and returns a reference to the local trace function to be used in that scope. So whenever any line is hit, it is reported by the tracing function which we could catch in our dictionary to store key value pairs of statements hit against the particular test case that was being run. We preferred settrace method over the python library coverage.py since we could directly use the coverage information given by settrace method inside our script, whereas for coverage it would have been an indirect method to first run on the command line and then feed the results to our script.

A basic arithmetic function then used the values stored in this dictionary datastructure to evaluate the suspiciousness score of each line while also taking into account whether that particular test failed failed or passed i.e. by measuring the number of passed test cases hit and the number of failed test cases hit by each line of code then pushing these as arguments to the suspiciousness function to calculate each line of code's score. The score may vary from 0 to 1 (1 being the most suspicious).

We wrote unit tests against three simple python programs against which our program was evaluated. Python's unittest library was used to test the programs and generate test data. For example, we wrote a mid() function that takes three numbers as input and should return the median of the three values as output. Our program successfully determined line 7 as the most suspicious statement where the bug was expected to be found and present. We also used Python's hypothesis

library to compute test cases which fail the program. This was particularly helpful because we were able to derive the test cases which would not meet the assertion and hence saved our time from manually figuring out the cases.

The ranking algorithm simply takes the suspiciousness values of all the lines of code and sorts them in descending order based on their suspiciousness score. Also, much effort was spent on automating our tool such that any python program with an arbitrary number of arguments, in the form of list or doubles or ints ,and an arbitrary number of test cases can be processed by it.

Program Repair

In the second part of PyFix, the line ranked highest by Tarantula is then passed on to the program repair module of PyFix.

Dividing script into distinct code blocks

Our aim, then, is to repair that buggy line considering the structure of code present within the buggedProg script. PyFix then attempts to divide the script into multiple distinct blocks of code the size of which is defined by the programmer in odd numbers e.g. 3 lines, 7 lines etc. For instance, considering the example that has been mentioned in this report of mid.py, PyFix divided the function into multiple 3-line blocks and ended up with a total of 11 blocks.

Computing Similarity of code blocks with bugged block

Afterwards, we compare the similarity of the bugged block (i.e. that specific block which contains the buggy line) with every other block of code. The goal here is to find a block of code inside buggedProg script with the highest similarity with our bugged block.

For measuring similarity, we used the “Levenshtein distance” method of the Python’s NLTK library which does a simple string metric measure between the two blocks of code which for the moment are treated as a string object and returns the percentage similarity between them. Informally, the Levenshtein distance between two strings is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one string into the other.[2] PyFix then picks the top three similar blocks into consideration for repairing the buggy line.

Repairing Code

For program repairing, PyFix follows a three-tiered approach that is

i) Permuting and mapping variables
ii) Permuting arithmetic operators
iii) Permuting bool statements

To begin with, first all the variables present within those three blocks are extracted using Python’s Abstract Syntax Trees parser[3]. Moving on, the variables found in the similar blocks are mapped with the variables present within the bugged line.

Along with that, if there are any operators present within the buggy line, (for example, ‘m=z+y’ has an assignment and addition operator) PyFix also attempts to map every other possible operator combination considering the context i.e. a programmer might have made a typo of assignment op

“=” instead of using a comparison “==” op but he is unlikely to have mistaken either of them for the mod “%” operator - and so mod would not be considered as a performance optimization. For permutations and combinations we used the Python’s itertools library. [4]

Moreover, keywords like ‘True’ or ‘False’ are also considered if our bugged line is a boolean statement such that “return True” will be replaced with “return False” and executed to see if all the testCases pass or not. This is relatively simpler in that all instances of True are replaced with False and vice versa.

All in all, PyFix attempts to map every possible combination of the variables and operators into the bugged line such that even a line as simple as “a = b + c” can have a plethora of combinations for instance:

a = c + b; b = a + c; b = c + a; c = a + b; c = b + a --- six combinations in a three variable statement without variable replacement

Given variables x, y, z
x = a + b; y = a + b; z = a + b --- three combinations when mapping only one variable

x = y + b; x = z + b; y = x + b; y = z + b; z = x + b; z = y + b --- six combinations when mapping only two variables

As you can see number of combinations will simply explode if we map more than three variables. Besides this we need to permute operators as well such that the following needs to be accounted for as well

a = b - c; a = b * c; a = b / c; a = b % c; a = b**c

Even though PyFix can handle an indefinite number of variables and operators but it would obviously become untenable due to time constraints especially when operators and variables are run in a nested configuration. This makes the benefit of blocks based similarity mapping very tangible as it would be unrealistic to permute over all the variables in say a 1000 line script.

Testing new script

Lastly, each time a change is made in the bugged line, PyFix replaces the bugged line in the original code which is then executed against all the given test cases which were initially passed as the third argument “testCases” to PyFix. It is important to check against both, previously passed and failed test cases, since a repaired code might have introduced new previously unknown or non-existent errors.

Therefore, every time a variant of the bugged script is generated it is simultaneously checked and in the instant any variant passes all the test cases, our program stops execution and returns the repaired version of code.

Getting Started

To run PyFix use the following command in terminal:
python PyFix.py <buggedProg> <testCaseResults> <testCases>

For example, you may try our given example as follows:

python PyFix.py mid.py testCasesMid testMid

```
def mid(x,y,z):
      m = z
      if (y<z):
            if(x<y):
                  m = y
            elif (x<z):
                  m = y
      else:
            if(x>y):
                  m = y
            elif (x>z):
                  m = x
      return m
```

Fig 1.1
mid.py

```
3,3,5
P
1,2,3
P
3,2,1
P
5,5,5
P
5,3,4
P
2,1,3
F
```

Fig 1.2
testCasesMid

```
def unittests(tempCodeString):
      try:
            exec(tempCodeString)
            assert mid(2,1,3) == 2
            assert mid(3,3,5) == 3
            assert mid(1,2,3) == 2
            assert mid(5,5,5) == 5
            assert mid(5,3,4) == 4
            assert mid(3,2,1) == 2
            # print 'Following Case
Passed:'
            return True
      except:
            # print "A test case
failed"
            return False
```

Fig 1.3
testMid.py

Case mid(2,1,3) in Fig 1.2 fails because it returns 1 instead of 2. This is resolved by PyFix

PyFix was written in Python 2.7 although it should work fine with Python 3.6 if the syntax changes in print() function is accounted for.

You would also need to install the following libraries:
i. NumPy
ii. NLTK

Checking the results

When run on mid.py, Tarantula gives the following output

```
//===-------------------- Running Tarantula ----------------------===//
Top 10 most suspicious lines
Line    Suspiciousness  Rank    Line of Code
7       0.833           1                       m = y
6       0.714           2               elif (x<z):
4       0.625           3               if(x<y):
1       0.5             7       def mid(x,y,z):
2       0.5             7               m = z
3       0.5             7               if (y<z):
13      0.5             7               return m
5       0.0             13                      m = y
8       0.0             13              else:
9       0.0             13                      if(x>y):
//===----------------------- Returns Tarantula -------------------------===//
//===--------------------    Bugged Line    ---------------------===//
m = y
```

Fig 2.1 TarantulaPy running mid.py

Here, the lines are listed in order of their likelihood of having a fault. In this case TarantulaPy successfully identifies that Line 7 has a bug.

The program repair module gives the following out when given the buggedLine as input from TarantulaPy

```
//===------------ Blocks: Start code with Bug Fix ---------------===//
def mid(x,y,z):
        m = z
        if (y<z):
                if(x<y):
                        m = y
                elif (x<z):
                        m = x
        else:
                if(x>y):
                        m = y
                elif (x>z):
                        m = x
        return m

//===------------- Blocks: End code with Bug Fix ----------------===//
```

Fig 2.2 Out from PyFix's program repair module when run on mid.py

In this case, line "m = y" has been replaced with "m = x" via Variable Mapping hence fixing the function.

Future Work/Limitations

- PyFix can as yet only repair bugs in a single line at a moment. In the case of multiple lines with bugs, TarantulaPy will be able to isolate the buggedLines but the repair module will be ineffective in debugging those lines simultaneously.
- PyFix was intended to be a very generic tool that works for all kinds of data types but we encountered a stupendous complication: In python, variables are dynamically typed and only values have concrete types. A variable x in a function can be a string at one point and in another line of code, it might be an instance of type int. There is no information of the type of variable before runtime. This essentially meant we could not just parse and extract the type of variable . Hence, we were forced to restrict PyFix on programs which contains variables of type int or float only. Type Checking remains a critical component that needs to be implemented.
- If there are problems present in the bugged program that cannot be corrected by just variation of operators and variables and bool statements, then PyFix will be unable to correct those issues given its current implementation.
- Machine Learning techniques could be applied to understand the code writing pattern of the user and based on that, intuit what could possibly cause the error. This would reduce the sample space and number of permutations currently performed, resulting in a significant deduction of processing time and allowing for larger scripts of code to be tested in reasonable time.

- PyFix has largely been tested on small python scripts spanning at most one file. We are not sure how our tool would perform on programs spanning multiple files and over thousands of lines of code.
- Unit test cases can be auto-generated against the given programs instead of being supplied and written by the user.
- For users' convenience a browser based interactive GUI could be developed that loads the test program and test cases into the client and displays them on an interactive screen.

Other Work

Mimicking KLEE Symbolic Execution Engine in Python

Source Code

https://github.com/ZainQasmi/KLEE-Symbolic-Execution-Py

What's KLEE?

KLEE is a program analysis tool based on symbolic execution and constraint solving techniques. It tries to find the possible inputs which will cause the program to crash or simply put, assertions to fail.
KLEE runs on LLVM bitcode files (.ll extension) generated by compiling C/C++ code using clang. STP constraint solver is used to produce the failing test cases.

Our Work in Python

We were inspired by the coverage and success rate of KLEE and wanted to replicate similar effects for Python. Since Python code could not be just compiled into LLVM bitcode, we decided to implement the tool in Python.

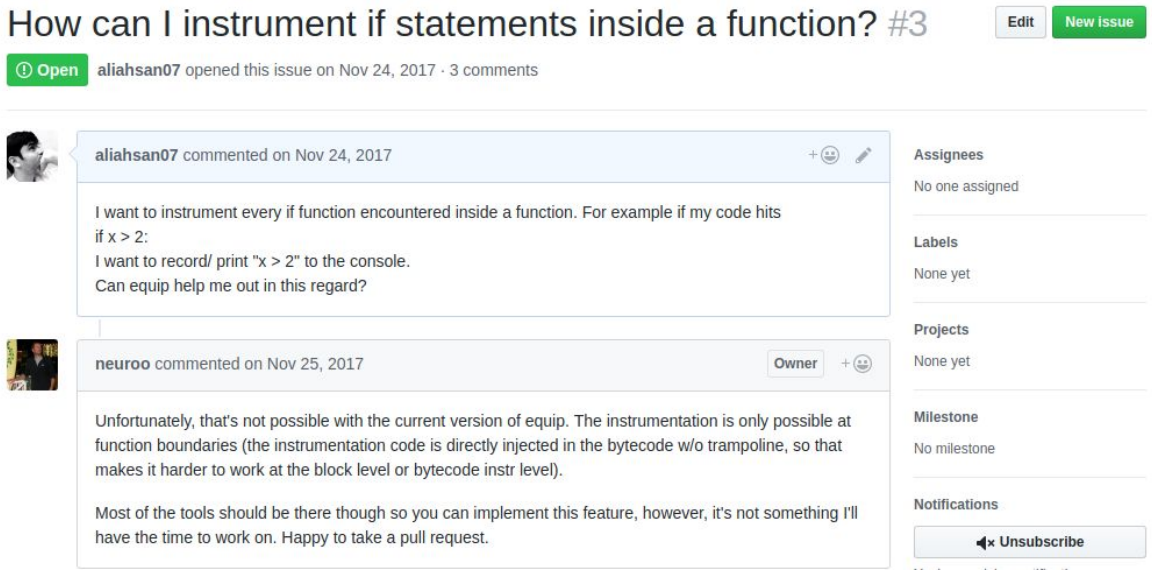
Our algorithm involved tracking the control flow graph of the code that is recording the path taken during the program(branches taken) and storing them. For example:

```
if x > 5 and y <0:
      # do something
      If x > 15:
            # do something
            # Assertion fails
      else:
            # do something
```

So the constraints recorded and stored for this example were `(x > 5 and y < 0)` and `(x > 15)`. These were then provided to Z3 theorem solver which would simplify the constraints to be `(x > 15 and y < 0)`. And it would output one such case i.e `x = 20` and `y = -20 { x : 20, y : -20 }`.

Why we had to forgo this Project

The problem arose when we had to instrument the Python code so that the paths could be stored and given to Z3 for evaluation. Unlike LLVM for C, there was nothing of that calibre for Python to instrument the code i.e record the branching conditions that were taken during the path.
One potential solution was using equip (an open source Python bytecode instrumentation) but that proved to be futile since equip only works on function boundaries while we wanted something to work on instruction level. We opened a relevant issue on their github repo (which is still unresolved). A screenshot has been attached for relevance:



How can I instrument if statements inside a function? #3

Edit | New issue

Open **aliahsan07** opened this issue on Nov 24, 2017 · 3 comments

aliahsan07 commented on Nov 24, 2017

I want to instrument every if function encountered inside a function. For example if my code hits
if x > 2:
I want to record/ print "x > 2" to the console.
Can equip help me out in this regard?

neuroo commented on Nov 25, 2017 — Owner

Unfortunately, that's not possible with the current version of equip. The instrumentation is only possible at function boundaries (the instrumentation code is directly injected in the bytecode w/o trampoline, so that makes it harder to work at the block level or bytecode instr level).

Most of the tools should be there though so you can implement this feature, however, it's not something I'll have the time to work on. Happy to take a pull request.

Assignees
No one assigned

Labels
None yet

Projects
None yet

Milestone
No milestone

Notifications
Unsubscribe

Since instrumentation was a key feature in automation of our tool, we couldn't carry the project further and instead decided to focus on automated program repairing in Python.

References

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6035728&tag=1 [1]

https://en.wikipedia.org/wiki/Levenshtein_distance [2]

https://docs.python.org/2/library/ast.html [3]

https://docs.python.org/2/library/itertools.html [4]

https://ieeexplore.ieee.org/document/7582831/ [5]

http://www.iisc-seal.net/deepfix [6]

http://spideruci.org/papers/jones05.pdf

https://github.com/neuroo/equip

https://github.com/Z3Prover/z3