An Algorithm for Seed Selection in Fuzzing

Tairan Song (ts2ww)
University of Virginia
Charlottesville, the United States
ts2ww@virginia.edu

Peiyi Yang (py5yy)
University of Virginia
Charlottesville, the United States
py5yy@virginia.edu

ACM Reference Format:
Tairan Song (ts2ww) and Peiyi Yang (py5yy). 2018. An Algorithm for Seed Selection in Fuzzing. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/1122445.1122456

1 ABSTRACT

In this paper, we mainly illustrate our algorithm selecting seeds for any fuzzing program. Our motivation is that current trending seed selection algorithms like Peach Set and Random Set do not take what paths the seed could potentially reach into consideration. We believe seeds with the most potential branches to explore have the largest chance to find bugs because much more unexplored paths could be reached. Therefore, given the call graph for the program under testing, we implement our algorithm by counting collective branch counts from current method and branch counts from all its children methods. To test the performance of our approach, we also implement Random Set algorithm and Peach Set algorithm as our baselines and compare the result of all these algorithms. We choose to use pythonfuzz as our fuzzer and use beautifulsoup4, pydotplus and pdfminer as our testing programs. In order to evaluate the performances of all three algorithms from different perspectives, we considered different conditions, such as different total number of seeds, different number of seeds input and different amount of time to run pythonfuzz. As indicated in results section, all three algorithms show stronger performance as the number of seeds increases. But the main advantage of our algorithm over Random Set and Peach Set is that our algorithm is more tolerant to the number of seeds given.

2 INTRODUCTION

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.[6] A mutation-based fuzzer leverages an existing corpus of seed inputs during fuzzing. It generates inputs by modifying (or rather mutating) the provided seeds. For example, when testing the Python program beautifulsoup4, we prepared 100 urls as seeds and input them into pythonfuzz. we noticed seeds are very important for generating good results from fuzzers as we were doing our homework 1 when we saw that different kinds of c programs as seeds can generate highly different coverage when they were input into AFL (American fuzzy lop). Some seeds were able to help AFL to reach a very high coverage in a few seconds. But some seeds caused AFL to be stuck at a certain level of coverage for a long time. Therefore, a good seed selection algorithm will improve a fuzzer's coverage significantly.
Current trending seed selection algorithms include[12]:

- PEACH SET: pick the top seeds according to coverage rate
- RANDOM SET : randomly pick seeds from seed library
- HOT SET : fuzz each seed for t seconds and pick the top k seeds by number of unique bugs found
- UNWEIGHTED MINSET: use an unweighted k-minset
- TIME MINSET : pick a k-execution time weighted minset
- SIZE MINSET : pick a k-size weighted minset. Many smaller files that cover a few code blocks may be preferable to one very large file that covers many code blocks.

However, all these algorithms are designed to select seeds to reach higher line coverage before inputting these seeds to fuzzer. In other words, they fail to take POTENTIAL branches into consideration. For example, suppose we have a program to be tested and Figure 1 is the call graph generated from this program. Each circle denotes the methods in the program under testing. Each arrow starts from source function to destination function(if we have an arrow from function A pointing to function B, it means function A calls function B). the number in each circle denotes the line coverage for a specific seed. If we have a seed S1 which will start from main function and goes to function A and we have another seed S2 which will go to the main function, function B and function C. If Peach Set is chosen as our seed selection algorithm which will pick the top seeds according to their line coverage. S2 will be picked over S1 since S2 has total coverage 4 but S2 has only 2. However, is S2 really better than S1? the answer is actually no. Even though S2 has a higher line coverage than S1, it does not have the potential to reach more coverage. For S1, it is highly possible that S1 could unlock all methods under function A as variable x be changed to True (which can be easily done by mutation in AFL). As we can see high coverage seeds do not always bring us a better chance of exploring unexplored parts in the program. Instead, seeds with the highest POTENTIAL to reach more unexplored paths could help fuzzer to find more bugs in more areas.

Therefore, in our project, we want to build a seed selection algorithm to focus on the POTENTIAL unexplored branches that the seeds could provide instead of focusing on the amount of coverage the seed has given us before we do fuzzing. For example, in Figure 2, suppose we have the same call graph for the same program.

Woodstock '18, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

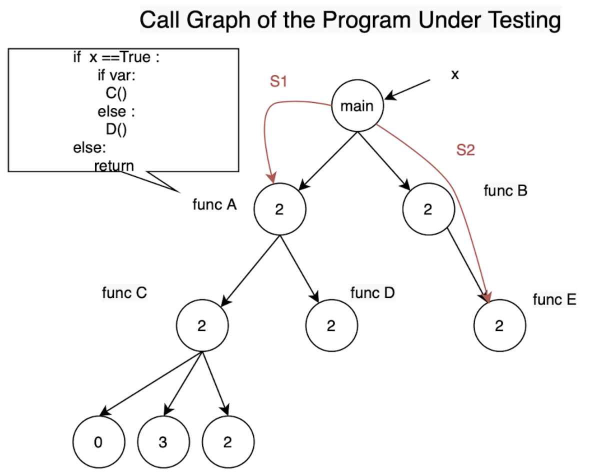


Figure 1: Call graph with peach set algorithm

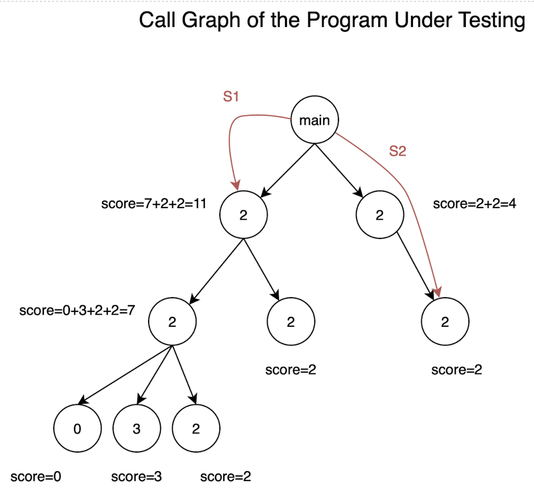


Figure 2: Call graph with our new algorithm

But for this time, we replace the numbers in each circle with the accumulative branch count for the current method and its children methods. For example, S2 łhave a score of 6 because it traverses two methods which have 2 branches and collectively 4 branches respectively. Similarly, S1 will have a collective score of 25 since S1 is given a score reflecting the sum of scores from all potential children methods it could reach. As we can see, S1 is more favorable than S2 in our new algorithm as we take potential branches into consideration.

3 APPROACH

3.1 Tools Used

Here's a list of existing tools that we used. One important assumption is that these tools are either bug-free, or no errors happened when we used them.

- pythonfuzz[6], a coverage guided fuzzer for testing python packages
- pyan[11], a call graph generator in the format of a dot file.
- pydot[8], a dot file parser and an interface to Graphviz.
- coverage.py[2], a tool for measuring code coverage of Python programs.
- crawler[5], a web crawler used to collect html files.

3.2 Workflow

Before we explain the workflow of our approach in detail, another assumption must be made: the libraries that are used by the tested programs are either bug-free, or no errors happened when we used them.

(1) (Optional) Pre-process the tested program.
(2) Collect seeds.
(3) Calculate the branch count of each function.
(4) Generate the call graph of the tested program.
(5) Calculate the score of each node in the call graph. For each function in the graph, its score equals to its branch count plus the sum of the scores of its children.
(6) Trace each seed: Run each seed on the tested program and collect its trace, which is a list of functions that were called.
(7) Calculate seed scores: For each seed, its score equals to the sum of scores of the functions in its trace.
(8) Select Seeds: Sort all seeds based on its seed score. Select the top N% seeds, where N is a hyper-parameter.
(9) Fuzz: Fuzz the tested program using the selected seed.
(10) Evaluate: Described in the next section.

3.3 Example: beautifulsoup4

In this section we will use beatifulsoup4 as an example and perform the workflow mentioned above. The package is install at /usr/local/lib/python3.8/dist-packages/bs4, and its structure is shown in Figure 4. Note that we are ignoring the __pychace__ folder, the tests folder and testing.py. Therefore, we'll be testing the following 9 .py files:

- /usr/local/lib/python3.8/dist-packages/bs4/__init__.py
- /usr/local/lib/python3.8/dist-packages/bs4/dammit.py
- /usr/local/lib/python3.8/dist-packages/bs4/diagnose.py
- /usr/local/lib/python3.8/dist-packages/bs4/element.py
- /usr/local/lib/python3.8/dist-packages/bs4/formatter.py
- /usr/local/lib/python3.8/dist-packages/bs4/builder/__init__.py
- /usr/local/lib/python3.8/dist-packages/bs4/builder/_lxml.py
- /usr/local/lib/python3.8/dist-packages/bs4/builder/_html5lib.py
- /usr/local/lib/python3.8/dist-packages/bs4/builder/_htmlparser.py

Step 1: (Optional) Pre-process the tested program

In the example of bs4 and throughout our experiment, the only

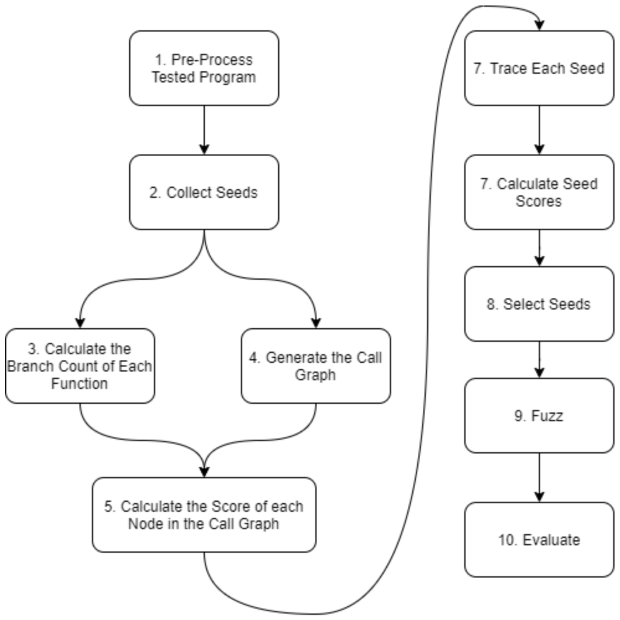
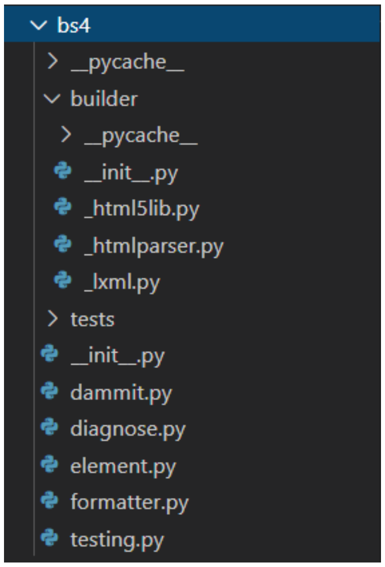

1. Pre-Process Tested Program
2. Collect Seeds
3. Calculate the Branch Count of Each Function
4. Generate the Call Graph
5. Calculate the Score of each Node in the Call Graph
7. Trace Each Seed
7. Calculate Seed Scores
8. Select Seeds
9. Fuzz
10. Evaluate

Figure 3: Workflow of our approach



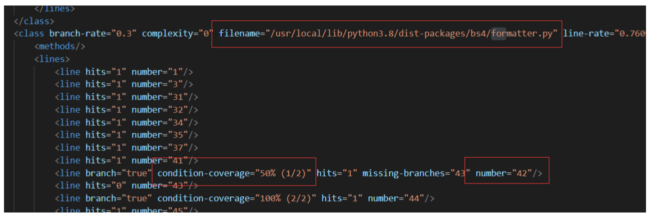
```
bs4
  __pycache__
  builder
    __pycache__
    __init__.py
    _html5lib.py
    _htmlparser.py
    _lxml.py
  tests
  __init__.py
  dammit.py
  diagnose.py
  element.py
  formatter.py
  testing.py
```

Figure 4: Structure of beautifulsoup4

modification we found that is needed is to remove all relative imports in the source code, i.e. replace "from .element import *" with "from bs4.element import *".

Step 2: Collect seeds
We used a crawler to crawl .html files from the web and use them as seeds.

Step 3: Calculate the branch count of each function
This step has 3 sub-steps. First, for every one of the 9 .py files, run:
coverage run –branch-count [.py file]
coverage xml
mv coverage.xml [.py file name].xml
Next, Process these .xml files. Figure 5 shows a generated .xml file. This .xml file tells us how many branches each line of the .py file contains (line 42 of formatter.py contains 2 branches). But what we really need is how many branches each function in that .py file contains. Therefore, we need to know the function name from the line number. This is achieved by parsing the .py file and building interval trees to record the start and end of each function and class, as is shown in Figure 6. With the help of interval trees, we will be able to find out that line 42 of formatter.py belongs to function _default of class Formatter.
We defined a simple rule for deciding the identifier for each function: [tested program]__[.py file name]__[[class name]__[function name], where a function can be nested inside other classes or functions, and the class name maybe be optional.
Following the rules and with some string manipulation, we will get "bs4_formatter__Formatter___default". Then, we put that identifier into a map and add its value by 2. Finally, for each .py file and .xml pair, perform the above process, and join their output map into a big score map.
Step 4: Generate the call graph



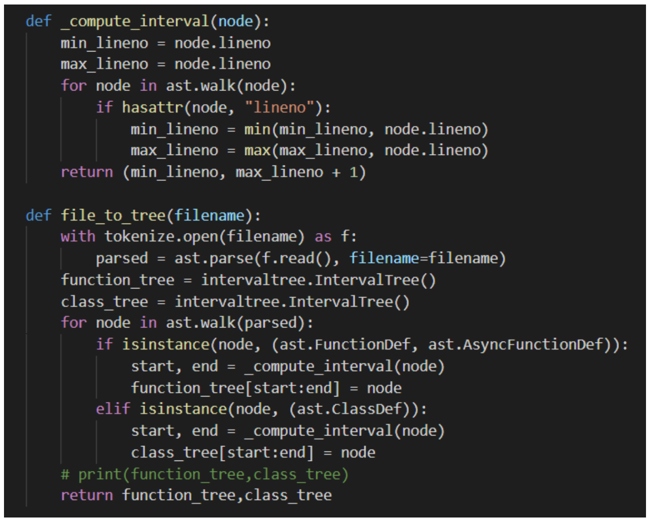
```
    </lines>
</class>
<class branch-rate="0.3" complexity="0" filename="/usr/local/lib/python3.8/dist-packages/bs4/formatter.py" line-rate="0.760
    <methods/>
    <lines>
        <line hits="1" number="1"/>
        <line hits="1" number="3"/>
        <line hits="1" number="31"/>
        <line hits="1" number="32"/>
        <line hits="1" number="34"/>
        <line hits="1" number="35"/>
        <line hits="1" number="37"/>
        <line hits="1" number="41"/>
        <line branch="true" condition-coverage="50% (1/2)" hits="1" missing-branches="43" number="42"/>
        <line hits="0" number="43"/>
        <line branch="true" condition-coverage="100% (2/2)" hits="1" number="44"/>
        <line hits="1" number="45"/>
```

Figure 5: Screenshot of a generated .xml file. This file tells us how many branch each line contains. In this example, line 42 of file formatter.py contains 2 branches.

In this step, first we generate the call graph of bs4 using the following command:
pyan [.py files] –uses –no-defines –colored –grouped –annotated –dot > bs4.dot
Next, process this .dot file. As shown in Figure 7, the generate .dot file contains node, which is functions and classes, and edges connecting the nodes. Next, convert all the node names into its standard identifier and build a call graph from these nodes and edges.

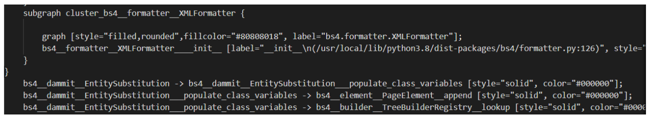
Step 5: Calculate the score of each node in the call graph
For each node in the call graph, use its standard identifier to find its branch count (if its identifier is not in the map, it means this node is a class or this node is a function that has no branches, so its branch count should be 0). Then recursively calculate its score, which equals to its branch count plus the sum of it's children's scores, as in shown in Figure 8.

Step 6: Trace Each Seed
Use python trace to trace all methods that are called for each seed. Then write the reports to .txt files, as shown in Figure 9. Process



```python
def _compute_interval(node):
    min_lineno = node.lineno
    max_lineno = node.lineno
    for node in ast.walk(node):
        if hasattr(node, "lineno"):
            min_lineno = min(min_lineno, node.lineno)
            max_lineno = max(max_lineno, node.lineno)
    return (min_lineno, max_lineno + 1)

def file_to_tree(filename):
    with tokenize.open(filename) as f:
        parsed = ast.parse(f.read(), filename=filename)
    function_tree = intervaltree.IntervalTree()
    class_tree = intervaltree.IntervalTree()
    for node in ast.walk(parsed):
        if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)):
            start, end = _compute_interval(node)
            function_tree[start:end] = node
        elif isinstance(node, (ast.ClassDef)):
            start, end = _compute_interval(node)
            class_tree[start:end] = node
    # print(function_tree,class_tree)
    return function_tree,class_tree
```

Figure 6: Building the interval tree for one .py file. We use the tree to find the function name given a line number.



```
    subgraph cluster_bs4__formatter__XMLFormatter {

        graph [style="filled,rounded",fillcolor="#80808018", label="bs4.formatter.XMLFormatter"];
        bs4__formatter__XMLFormatter___init__ [label="__init__\n(/usr/local/lib/python3.8/dist-packages/bs4/formatter.py:126)", style="
    }
}
    bs4__dammit__EntitySubstitution -> bs4__dammit__EntitySubstitution___populate_class_variables [style="solid", color="#000000"];
    bs4__dammit__EntitySubstitution___populate_class_variables -> bs4__element__PageElement__append [style="solid", color="#000000"];
    bs4__dammit__EntitySubstitution___populate_class_variables -> bs4__builder__TreeBuilderRegistry__lookup [style="solid", color="#0000
```

Figure 7: Screenshot of bs4.dot file. Above: nodes (functions/classes), below: edges. Note that the node names differ from our defined identifier rule and needs processing.

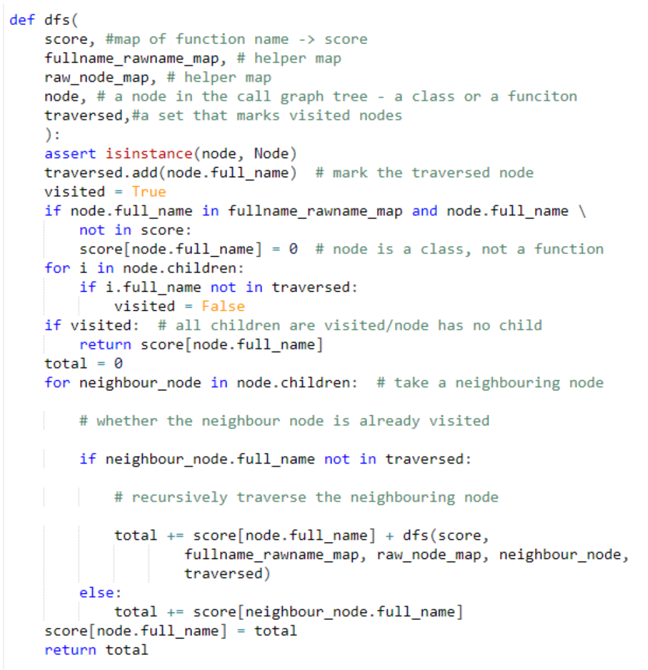
these file, extract methods that are inside bs4 site-package.

Step 7: Calculate seed scores
Given the dictionary recording scores for each method and the total path (all methods) that each seed traverses, we calculate a map having keys in format of [tested program name]__[.py file name]__[[(optional)class name]__[function name] and values as the cumulative scores for each seed, as shown in Figure 10.

Step 8: Select Seeds
We select top 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% of all seeds for each algorithm for comparison. For Random set, we just pick these percentages of files randomly.

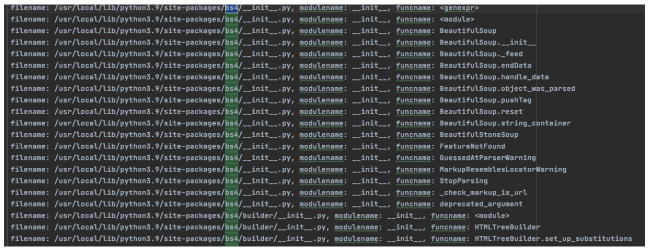
Step 9: Fuzz
For each percentage of seeds in each seed selection algorithm, we input the seeds in each folder shown in Figure 11 to pythonfuzz. We set fuzzing time for each seed group to 30 seconds. We also tried fuzzing for 1 minute. But the results for 1 minutes turned out to be roughly the same (each seed group reaches to about the same coverage).

Step 10: Evaluate
Please see evaluation section.



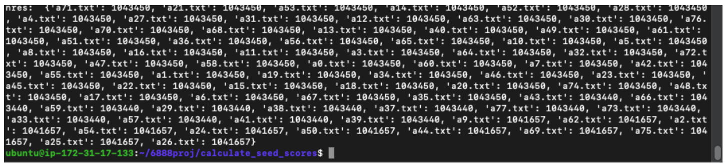
```python
def dfs(
    score, #map of function name -> score
    fullname_rawname_map, # helper map
    raw_node_map, # helper map
    node, # a node in the call graph tree - a class or a funciton
    traversed,#a set that marks visited nodes
    ):
    assert isinstance(node, Node)
    traversed.add(node.full_name)  # mark the traversed node
    visited = True
    if node.full_name in fullname_rawname_map and node.full_name \
        not in score:
        score[node.full_name] = 0  # node is a class, not a function
    for i in node.children:
        if i.full_name not in traversed:
            visited = False
    if visited:  # all children are visited/node has no child
        return score[node.full_name]
    total = 0
    for neighbour_node in node.children:  # take a neighbouring node

        # whether the neighbour node is already visited

        if neighbour_node.full_name not in traversed:

            # recursively traverse the neighbouring node

            total += score[node.full_name] + dfs(score,
                    fullname_rawname_map, raw_node_map, neighbour_node,
                    traversed)
        else:
            total += score[neighbour_node.full_name]
    score[node.full_name] = total
    return total
```

Figure 8: Calculating the score of each function



```
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: <genexpr>
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: <module>
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: BeautifulSoup
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: BeautifulSoup.__init__
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: BeautifulSoup._feed
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: BeautifulSoup.endData
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: BeautifulSoup.handle_data
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: BeautifulSoup.object_was_parsed
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: BeautifulSoup.pushTag
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: BeautifulSoup.reset
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: BeautifulSoup.string_container
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: BeautifulStoneSoup
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: FeatureNotFound
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: GuessedAtParserWarning
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: MarkupResemblesLocatorWarning
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: StopParsing
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: _check_markup_is_url
filename: /usr/local/lib/python3.9/site-packages/bs4/__init__.py, modulename: __init__, funcname: deprecated_argument
filename: /usr/local/lib/python3.9/site-packages/bs4/builder/__init__.py, modulename: __init__, funcname: <module>
filename: /usr/local/lib/python3.9/site-packages/bs4/builder/__init__.py, modulename: __init__, funcname: HTMLTreeBuilder
filename: /usr/local/lib/python3.9/site-packages/bs4/builder/__init__.py, modulename: __init__, funcname: HTMLTreeBuilder.set_up_substitutions
```

Figure 9: Trace of functions/methods called



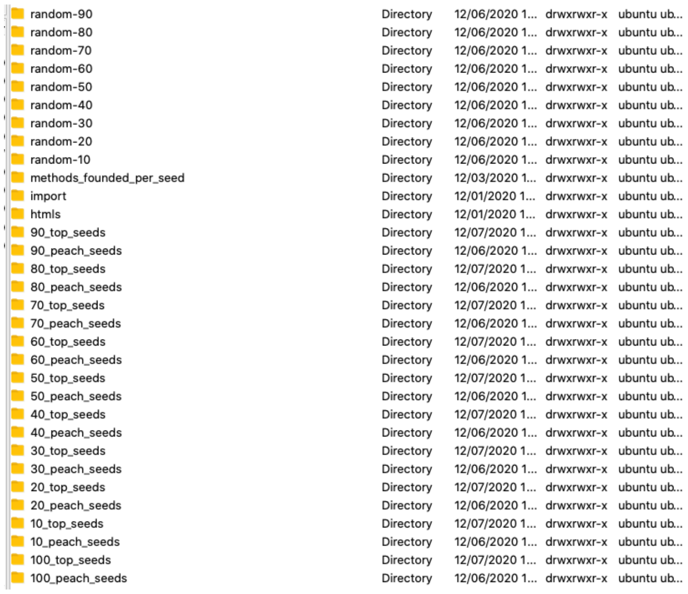
```
nres:  {'a71.txt': 1043450, 'a21.txt': 1043450, 'a53.txt': 1043450, 'a14.txt': 1043450, 'a52.txt': 1043450, 'a28.txt': 1043450
, 'a4.txt': 1043450, 'a27.txt': 1043450, 'a31.txt': 1043450, 'a12.txt': 1043450, 'a63.txt': 1043450, 'a30.txt': 1043450, 'a76.
txt': 1043450, 'a70.txt': 1043450, 'a68.txt': 1043450, 'a13.txt': 1043450, 'a40.txt': 1043450, 'a49.txt': 1043450, 'a61.txt':
1043450, 'a51.txt': 1043450, 'a36.txt': 1043450, 'a56.txt': 1043450, 'a65.txt': 1043450, 'a10.txt': 1043450, 'a5.txt': 1043450
, 'a8.txt': 1043450, 'a16.txt': 1043450, 'a11.txt': 1043450, 'a3.txt': 1043450, 'a64.txt': 1043450, 'a32.txt': 1043450, 'a72.t
xt': 1043450, 'a47.txt': 1043450, 'a58.txt': 1043450, 'a0.txt': 1043450, 'a60.txt': 1043450, 'a7.txt': 1043450, 'a42.txt': 104
3450, 'a55.txt': 1043450, 'a1.txt': 1043450, 'a19.txt': 1043450, 'a34.txt': 1043450, 'a46.txt': 1043450, 'a23.txt': 1043450, '
a45.txt': 1043450, 'a22.txt': 1043450, 'a15.txt': 1043450, 'a18.txt': 1043450, 'a20.txt': 1043450, 'a74.txt': 1043450, 'a48.tx
t': 1043450, 'a17.txt': 1043450, 'a6.txt': 1043450, 'a67.txt': 1043450, 'a35.txt': 1043450, 'a43.txt': 1043440, 'a66.txt': 104
3440, 'a59.txt': 1043440, 'a29.txt': 1043440, 'a38.txt': 1043440, 'a37.txt': 1043440, 'a77.txt': 1043440, 'a73.txt': 1043440,
'a33.txt': 1043440, 'a57.txt': 1043440, 'a41.txt': 1043440, 'a39.txt': 1043440, 'a9.txt': 1041657, 'a62.txt': 1041657, 'a2.txt
': 1041657, 'a54.txt': 1041657, 'a24.txt': 1041657, 'a50.txt': 1041657, 'a44.txt': 1041657, 'a69.txt': 1041657, 'a75.txt': 104
1657, 'a25.txt': 1041657, 'a26.txt': 1041657}
ubuntu@ip-172-31-17-133:~/6888proj/calculate_seed_scores$
```

Figure 10: Seed scores.

4 EVALUATION

4.1 Programs Tested

We tested several open source Python packages listed in Table 1. During our experiment, we found that for some programs, the results given by three different seed selection algorithms were very close, namingly, jsonparser and pycallgraph2. We figured it was because these programs themselves are too small. Therefore no matter what seed it is fed with, the traces are almost identical. As a result, we only evaluate the results of bs4, pydotplus and pdfminer.six, and disregard the others.



```
random-90              Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
random-80              Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
random-70              Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
random-60              Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
random-50              Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
random-40              Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
random-30              Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
random-20              Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
random-10              Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
methods_founded_per_seed  Directory  12/03/2020 1...  drwxrwxr-x  ubuntu ub...
import                 Directory  12/01/2020 1...  drwxrwxr-x  ubuntu ub...
htmls                  Directory  12/01/2020 1...  drwxrwxr-x  ubuntu ub...
90_top_seeds           Directory  12/07/2020 1...  drwxrwxr-x  ubuntu ub...
90_peach_seeds         Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
80_top_seeds           Directory  12/07/2020 1...  drwxrwxr-x  ubuntu ub...
80_peach_seeds         Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
70_top_seeds           Directory  12/07/2020 1...  drwxrwxr-x  ubuntu ub...
70_peach_seeds         Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
60_top_seeds           Directory  12/07/2020 1...  drwxrwxr-x  ubuntu ub...
60_peach_seeds         Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
50_top_seeds           Directory  12/07/2020 1...  drwxrwxr-x  ubuntu ub...
50_peach_seeds         Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
40_top_seeds           Directory  12/07/2020 1...  drwxrwxr-x  ubuntu ub...
40_peach_seeds         Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
30_top_seeds           Directory  12/07/2020 1...  drwxrwxr-x  ubuntu ub...
30_peach_seeds         Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
20_top_seeds           Directory  12/07/2020 1...  drwxrwxr-x  ubuntu ub...
20_peach_seeds         Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
10_top_seeds           Directory  12/07/2020 1...  drwxrwxr-x  ubuntu ub...
10_peach_seeds         Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
100_top_seeds          Directory  12/07/2020 1...  drwxrwxr-x  ubuntu ub...
100_peach_seeds        Directory  12/06/2020 1...  drwxrwxr-x  ubuntu ub...
```

Figure 11: 10-90 percent of seeds per algorithm

Package Name	Size (LOC)	Seed Format	Used	Seed Number
beautifulsoup4[1]	medium	.html files, text	yes	78
pydotplus[4]	small	.dot files, text	yes	202
pdfminer.six[7]	large	.pdf files, binary	yes	15
jsonparser[9]	small	.json files, text	no	N/A
pycallgraph2[10]	small	.py files, text	no	N/A

Table 1: Comparison of programs that are tested.

4.2 Baselines

Two algorithms are used as the baselines:

(1) **Random** In the Random algorithm, we simply select N% seeds randomly.
(2) **Peach** We referred to the seed selection algorithm in the popular fuzzer Peach[3] and implemented a simple version on our own, where we first sort the seeds based on their line coverage, and then select the top N%.

4.3 Original Design: Based on Number of Bugs Found

The ultimate goal is to find the subset of seeds that trigger the most unique bugs in a program, given a run-time threshold. During runtime the following information will be collected:

- The structure tree of the program being tested.
- The execution path of a seed file.
- Lines of code that are related to a crash. We use a hash function on the crash-related lines to automatically determine if a bug is unique.
- The timestamp when a bug is found.

Based on the above information, we can calculate the number of unique bugs found and the time cost to find each bug, and turn our problem into an integer linear program problem using the constraints in [12], to measure the quality of a seed set, regardless of the specific scheduling algorithm and fuzzer: First, define an indicator variable b

$$b_x = \begin{cases} 1 & \text{bug x is unique} \\ 0 & \text{otherwise} \end{cases}$$

Define an indicator variable c

$$b_{i,j} = \begin{cases} 1 & \text{the schedule includes crash j for seed i} \\ 0 & \text{otherwise} \end{cases}$$

We introduce a hash function to determine if two crashes are essentially the same bug:

$$b_x = 1 iff \forall i,j : hash(c_{i,j}) = x$$

The time cost for each crash:

$$\forall t_{i,j} = \begin{cases} a_{i,1} & \text{j=1} \\ a_{i,j} - a_{i,j-1} & \text{otherwise} \end{cases}$$

And the ILP problem is formulated as follows:

$$\text{maximize} \sum_x b_x$$

$$\text{subject to}$$

$$\forall_{i,j}, c_{i,j+1} \leq c_{i,j}$$

$$\sum_{i,j} c_{i,j} t_{i,j} \leq t_{thres}$$

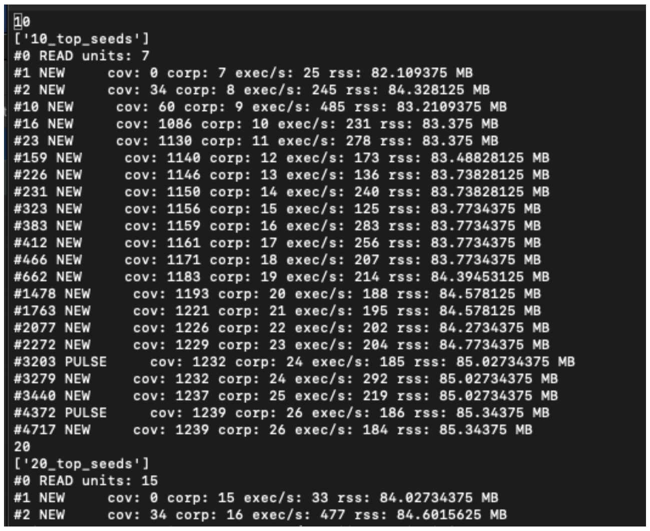
$$\forall_{i,j}, c_{i,j} \leq b_x where hash(c_{i,j} = x)$$

$$\forall_x b_x \leq \sum_{i,j} c_{i,j} where hash(c_{i,j} = x)$$

Furthermore, we also consider the transferability of seed files. Determining a "good" seed set for a particular program can be time-consuming, as we need to run the program multiple times with every seed in the seed set, across all seed sets. Instead, once we found a good seed set, we would want to reuse it on other similar programs.

4.4 Re-design: Based on Coverage

The previous evaluation relies on the assumption that the tested program contains bugs, and that at least one bug will lead to an crash. However, as much as we would like to assume so, we really don't know if it the above two conditions are true and can not assume there will be crashes. That requires us to use another metric to evaluate our approach: coverage. Here we made an assumption: seeds with higher coverage are better than seeds with lower coverage, because if the tested program contains bugs, they are more likely to cause a crash.
Figure 12 shows the output of pythonfuzz, where each row is one round of fuzzing and the 3rd column "cov:number" means the line coverage achieved at that round. Our new design is as follows: run each seed selection algorithm, collect the coverage achieved at the last round of fuzzing as the metric, and whichever algorithm achieves higher coverage is considered better with respect to the fixed time T and percentage of seeds used N.



```
10
['10_top_seeds']
#0 READ units: 7
#1 NEW     cov: 0 corp: 7 exec/s: 25 rss: 82.109375 MB
#2 NEW     cov: 34 corp: 8 exec/s: 245 rss: 84.328125 MB
#10 NEW     cov: 60 corp: 9 exec/s: 485 rss: 83.2109375 MB
#16 NEW     cov: 1086 corp: 10 exec/s: 231 rss: 83.375 MB
#23 NEW     cov: 1130 corp: 11 exec/s: 278 rss: 83.375 MB
#159 NEW     cov: 1140 corp: 12 exec/s: 173 rss: 83.48828125 MB
#226 NEW     cov: 1146 corp: 13 exec/s: 136 rss: 83.73828125 MB
#231 NEW     cov: 1150 corp: 14 exec/s: 240 rss: 83.73828125 MB
#323 NEW     cov: 1156 corp: 15 exec/s: 125 rss: 83.7734375 MB
#383 NEW     cov: 1159 corp: 16 exec/s: 283 rss: 83.7734375 MB
#412 NEW     cov: 1161 corp: 17 exec/s: 256 rss: 83.7734375 MB
#466 NEW     cov: 1171 corp: 18 exec/s: 207 rss: 83.7734375 MB
#662 NEW     cov: 1183 corp: 19 exec/s: 214 rss: 84.39453125 MB
#1478 NEW     cov: 1193 corp: 20 exec/s: 188 rss: 84.578125 MB
#1763 NEW     cov: 1221 corp: 21 exec/s: 195 rss: 84.578125 MB
#2077 NEW     cov: 1226 corp: 22 exec/s: 202 rss: 84.2734375 MB
#2272 NEW     cov: 1229 corp: 23 exec/s: 204 rss: 84.7734375 MB
#3203 PULSE     cov: 1232 corp: 24 exec/s: 185 rss: 85.02734375 MB
#3279 NEW     cov: 1232 corp: 24 exec/s: 292 rss: 85.02734375 MB
#3440 NEW     cov: 1237 corp: 25 exec/s: 219 rss: 85.02734375 MB
#4372 PULSE     cov: 1239 corp: 26 exec/s: 186 rss: 85.34375 MB
#4717 NEW     cov: 1239 corp: 26 exec/s: 184 rss: 85.34375 MB
20
['20_top_seeds']
#0 READ units: 15
#1 NEW     cov: 0 corp: 15 exec/s: 33 rss: 84.02734375 MB
#2 NEW     cov: 34 corp: 16 exec/s: 477 rss: 84.6015625 MB
```

Figure 12: pythonfuzz output

time=30s	program=bs4	seed=78	
top N%	random(baseline)	peach	top(our Approach)
10	1255	1243	1239
20	1290	1241	1239
30	1762	1272	1278
40	1764	1752	1754
50	1767	1762	1778
60	1756	1757	1758
70	1765	1299	1759
80	1307	1767	1741
90	1761	1762	1313

Table 2: top-random-peach results for program bs4.

time=30s	program=pydotplus		seed=202
top N%	random(baseline)	peach	top(our Approach)
10	**2175**	**2183**	2189
20	2173	**2206**	2187
30	2146	**2215**	2193
40	2186	2123	2170
50	2156	2134	2181
60	2137	2176	2176
70	2179	2180	2189
80	2183	2180	2180
90	2180	2180	2180

Table 3: top-random-peach results for program pydotplus.

time=30s	program=pdfminer.six		seed=15
top N%	random(baseline)	peach	top(our Approach)
10	744	757	925
20	878	914	958
30	951	924	971
40	997	984	959
50	995	970	965
60	1170	1002	1032
70	1174	1147	1034
80	1163	1168	1153
90	1171	1171	1191

Table 4: top-random-peach results for program pdfminer.

4.5 Results

4.5.1 bs4. Table 2 and Figure 13 shows the results of testing bs4. Surprisingly, pythonfuzz did not find any crashes for all seed groups. However, each seeds group has different performance in terms of coverage. As we can see Figure 13, Random has a generally better performance when the number of seeds is small (20% - 50% of total seeds(78)). Also, the performance of our approach(top) is good when we input 40% of total seeds and the result shows that the coverage generated by our algorithm is more flexible than the coverage generated from the other two algorithms. It does not vary much for different number of seeds. However, the coverage generated by Random selection and Peach set algorithm are good only for a certain range of seed numbers (30% - 70% for Random and 40% - 60% for Peach Set).

4.5.2 pydotplus. Table 3 and Figure 14 show the results of testing pydotplus, where bold fonts indicate crashes identified by pythonfuzz. This time, because the total number of seeds are quite large (202) and pydotplus itself is a small-size program, the choice of N didn't make much difference in terms of coverage. However, when N was small, Peach and Random was able to make the program crash. Although our approach (top) didn't find any crashes, it was able to reach the highest coverage, 2193 when N=30, while Peach can not go beyond 2180 regardless of N.

4.5.3 pdfminer. Table 4 and Figure 15 show the results of testing pdfminer. As we can see from Figure 15, the coverage generated from Random and Peach start with low numbers and increase gradually as more seeds are input. But the coverage for top(our approach) starts with a higher number and maintain this number for a while and then jumps when 70% of seeds are input. Overall, all these three algorithm are not affected by the number of seeds very much (coverage 750-1200). Top(our approach) has the best performance when only a few seeds are given. However, top does not have good improvement over time compared to the other two algorithms. Finally, all three algorithms have very good improvement when about 65% of seeds are given.

5 CONCLUSIONS

Our major findings are:

(1) Generally speaking, with any seed selection algorithm, a coverage-guided fuzzer like pythonfuzz can not reach a satisfying coverage if we don't feed enough seeds into the fuzzer at the first place.
(2) The choice of hyper-parameter N: there isn't a value that's universally good for every program being tested. But approximate range should be 50 90. Also, it is not true that a larger

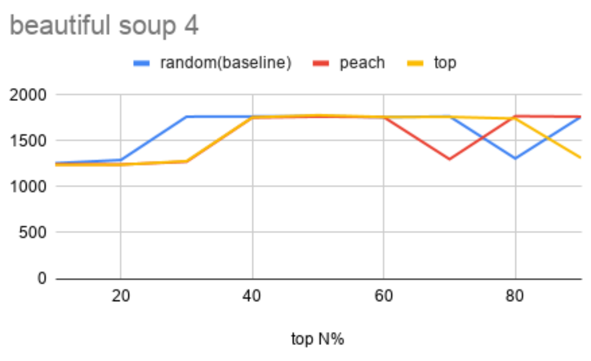


Figure 13: beautifulsoup4 result

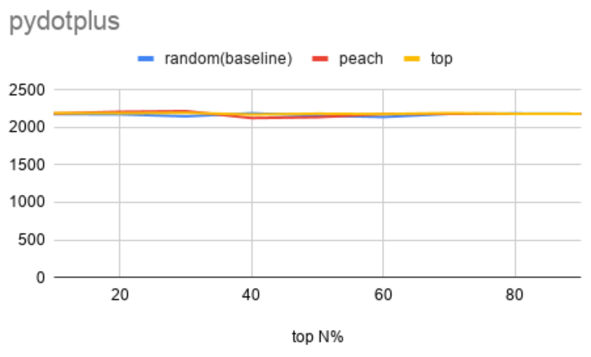


Figure 14: pydotplus result

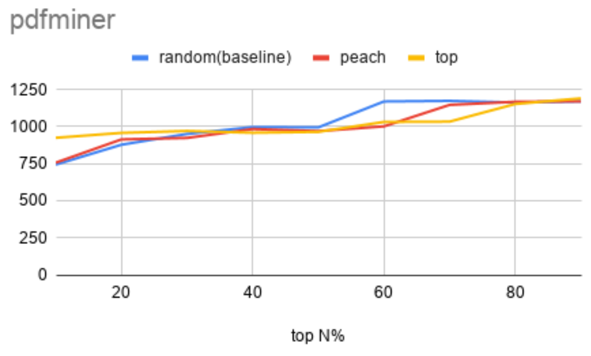


Figure 15: pdfminer result

N will always result in a higher coverage. In contrast, using a large N will not help the fuzzer achieve a higher coverage and in some cases will cause the coverage to drop.

(3) Our proposed approach is slightly better than Peach because it is more tolerant of N and gives more stable results. Peach is slightly better than Random. pythonfuzz is not as sophisticated as AFL as we find that pythonfuzz is usually not able to have fast discovery of new paths. Instead, it will stuck at some point when a certain threshold is reached. However, AFL is usually able to keep mutating seeds and digging into new paths quickly. More importantly, pythonfuzz provides us less detailed report than AFL. Therefore, we have to rely on coverage rate for measuring the performance of each algorithm when no crash is found.

There are some factors whose impact remain unknown to us:

(1) The choice of the fuzzer: we don't know if pythonfuzz itself is as sophisticated as other mature fuzzers, such as AFL, which has been developing for years. It is possible that pythonfuzz contains some bugs and gives misleading results.
(2) The reliability of pythonfuzz remains unknown. For example, as we test pdfminer, we run pythonfuzz for several times but find out the few results are not very consistent. Sometimes we are able to find out crashes but sometimes the crashes are not detected anymore.
(3) The size of the tested program: intuitively, larger programs are prune to more bugs but also a larger search space. But we don't know the exact relationship between the program size and testing results.
(4) The time cost of finding a bug: in our experiment, we only run pythonfuzz for a fixed time and use the coverage to evaluate our approach, because we don't know for sure that the tested programs actually contains bugs (and we didn't assume so). In that sense, We also don't know if running the tool for a much longer time (for hours or days) will give us crashes.

With the above limitations discussed, one can easily think of many aspects to further improve our work:

(1) Use different fuzzers as we notice pythonfuzz is not able to provide us a more detailed report
(2) Test more programs of different sizes. As we can see from the results of pydotPlus, small size of program testing could possibly give us similar result for every seed selection algorithms because all seeds are likely explore mostly same paths. Therefore, testing larger and complex programs will provide us more diverse results.
(3) Run for a longer time. For this project, we only run pythonfuzz for 30s as we find that results tend to be similar for longer time. But if we changed to better fuzzer and more seeds, having the fuzzer running for a longer time will help us to get better coverage or more bugs
(4) Run fuzzer for more times repeatedly. Due to time constraint, we only run pythonfuzz for each algorithm on each amount of seeds for 3 times and calculate the averages. We believe having pythonfuzz run repeatedly for more times (>10 times) could give us better insights.
(5) Generate larger amount of seeds. So far we don't have large amount of seeds (the most seeds we have is 202). In these small set of seeds, seeds could be very similar to each other which causes that many seeds could traverse to similar or exactly the same methods so that these seeds will be given the same scores. This situation happened when we are testing jsonparser and pyCallGraph2 as all seeds we find traverse exactly the same path in these programs. Therefore, larger

amount of seeds will not only give us diverse results but also help fuzzer to find explore more paths.

(6) Compare our algorithm to more sophisticated seed selection algorithms such as UNWEIGHTED MINSET, TIME MINSET and SIZE MINSET mentioned in this paper[12].

(7) We will be able to have a more detailed analysis if we could test more different amount of seeds per algorithm.

REFERENCES

[1] [n.d.]. beautifulsoup4: Screen-scraping library. http://www.crummy.com/software/BeautifulSoup/bs4/

[2] [n.d.]. Coverage.py — Coverage.py 5.3 documentation. https://coverage.readthedocs.io/en/coverage-5.3/

[3] [n.d.]. Peach Fuzzer. https://www.peach.tech/products/peach-fuzzer/

[4] [n.d.]. pydotplus: Python interface to Graphviz's Dot language. http://pydotplus.readthedocs.org/

[5] [n.d.]. –_-CSDN. https://blog.csdn.net/moluth/article/details/82906765

[6] 2020. fuzzitdev/pythonfuzz. https://github.com/fuzzitdev/pythonfuzz original-date: 2019-10-23T16:10:59Z.

[7] 2020. pdfminer/pdfminer.six. https://github.com/pdfminer/pdfminer.six original-date: 2014-08-29T14:04:53Z.

[8] 2020. pydot/pydot. https://github.com/pydot/pydot original-date: 2015-04-13T15:14:44Z.

[9] Leonid Bugaev. 2020. buger/jsonparser. https://github.com/buger/jsonparser original-date: 2016-03-20T06:22:33Z.

[10] Dan Eads. 2020. daneads/pycallgraph2. https://github.com/daneads/pycallgraph2 original-date: 2018-11-10T22:31:06Z.

[11] David Fraser. 2020. davidfraser/pyan. https://github.com/davidfraser/pyan original-date: 2016-01-20T16:18:44Z.

[12] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 861–875.