Overview of DPFuzz (Paper 50)

Saeid Tizpaz-Niari

University of Colorado Boulder

1 Overview of DPFuzz

DPFuzz consists of five steps: 1) Fuzzing, 2) Clustering, 3) Classification in the space of program inputs, 4) Instrumentations, and 5) Classification in the space of program internals. The steps 1 and 2 are the discovery phase and steps 2 to 5 are the explanation phase. Figure 1 shows the workflow of DPFuzz.

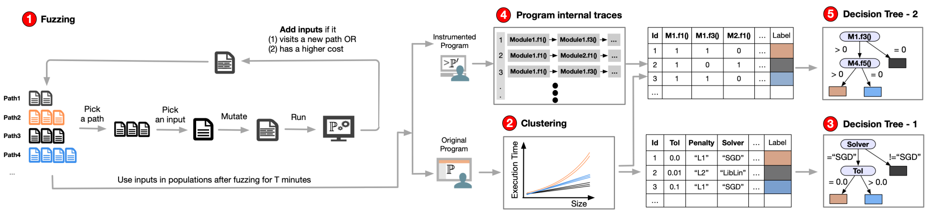
1) Fuzzing. The first component of DPFuzz is to generate inputs. For this part, we use an evolutionary fuzzing algorithm. Our fuzzing approach considers multiple populations (one population per a distinct path in the CFG). Then, it picks a cluster, a path from the cluster, and an input from the selected path. Next, it mutates and crossovers the input and runs the input on the target program. This returns the cost of executions (either in terms of actual execution times or in terms of executed lines), and the path characterization. The fuzzing approach adds a new input to the populations if the input has visited a new path in the program or the input has achieved higher costs in comparison to the inputs in the same path. The fuzzing stops after T time units and provides generated inputs for the next steps.

Highlight. This step discovers the interesting inputs to characterize different classes of performances and to analyze them further to determine if there is a performance bug.

2) Clustering. The second component of DPFuzz is to discover different classes of performances using the inputs from the fuzzing step. The set of inputs in a path (or a population) defines a performance function varied in the input size. Given n paths (corresponds to n performance functions), DPFuzz applies clustering algorithms to partition theses functions into k classes of performances ($k \leq n$). The clustering is primarily based on the non-parametric functional data clustering with l_1 distance. The clustering helps the user focus among a few classes of distinguishable performances, rather than numerous paths in the library, many of those may have similar performances.

Highlight. This step discovers distinguishable classes of performances and provide the performance class label for each input. These results are crucial for the explanation phase.

3) Classification in the input space. The third component of DPFuzz is to explain different performances in terms of program inputs and internals. The CART decision tree inference is mainly used to obtain the explanation models. In the space of program inputs, the features are input parameters such as the value of “solver” and the labels are the performance classes from the clustering algorithm. The decision tree (1) in Figure 1 is learned in the space of program



Id	M1.f1()	M1.f3()	M2.f1()	...	Label
1	1	1	0	...	🟫
2	1	0	1	...	⬛
3	1	1	0	...	🟦
...					

Id	Tol	Penalty	Solver	...	Label
1	0.0	"L1"	"SGD"	...	🟫
2	0.01	"L2"	"LibLin"	...	⬛
3	0.1	"L1"	"SGD"	...	🟦
...					

Fig. 1. DPFuzz workflow. DPFuzz consists of five steps: (1) fuzzing to generate interesting inputs, (2) clustering to find classes of performance functions, (3) classification in the space of input features (parameters of ML libraries), (4) instrumentations to generate ML libraries' internal features such as whether a function is called, and (5) classification in the space of internal features to localize (suspicious) code regions.

inputs that explain what input parameters are common in the same cluster and what the parameters distinguish one cluster from another. Using this decision tree, the user may realize that all or some aspects of differential performances are unexpected. For example, in Figure 1 (part 3), the differences between "Tol" = 0.0 and "Tol" $>$ 0.0 (such as "Tol" = 0.000001) under the same "solver" = SGD is unexpected and require further analyze to determine whether the differences are intrinsic to the problem or they are results of performance bugs.
Highlight. The classification in the space of program inputs aims to explain the root cause of performance differences in terms of library's input parameters. Once an unexpected performance is detected, the next step is to turn to program internals and localize code regions contributed to the unexpected performance classes.
4) Instrumentations. The fourth component is to instrument a given library and generate internal features such as a condition or function call. For this aim, tracing techniques from python `Trace` library are used. The idea is to run each input on the instrumented library and obtain the number of calls to "if", "for", "while" statements and calls to the functions for that input.
Highlight. The instrumentation step generates program internal features, and they are inputs for the classification in the space of program internals.
5) Classification in the internal space. Given the program internal features (from step 4) and the cluster label (from step 2), the CART decision tree inference is applied to explain different performance clusters based on program internal features. The explanation helps the user localize region in the code that most likely contribute to observe an unexpected performance behavior. For example, in Figure 1 (part 5), calls to the function `f5` in the module `M4` are the root cause for the performance differences. Now, the user can look further inside these code regions and determine whether the performance differences are intrinsic to the problem or they are results of a performance bug.
Highlight. The decision tree classifier with the features from the internals of a library pinpoints code regions that are most likely contributing for an unexpected performance class.