The Optimization of a Symbolic Execution Engine for Detecting Runtime Errors

István Kádár

Nowadays, producing great, reliable and robust software systems is quite a big challenge in software engineering. About 40% of the total development costs go for testing [1] and maintenance activities, moreover, bug fixing of the system also consumes a considerable amount of resources [2].

The symbolic execution engine called RTEHunter developed at the Department of Software Engineering at the University of Szeged supports this phase of the software engineering lifecycle by detecting runtime errors (such as null pointer dereference, bad array indexing, division by zero) in Java programs without actually running the program in real-life environment.

During its execution, every program performs operations on the input data in a defined order. Symbolic execution [3] is based on the idea that the program is operated on symbolic variables instead of specific input data, and the output will be a function of these symbolic variables. A symbolic variable is a set of the possible values of a concrete variable in the program, thus a symbolic state is a set of concrete program states. When the execution of the program reaches a branching condition containing a symbolic variable, the condition cannot be evaluated and the execution continues on both branches. The execution paths created this way compose a tree called symbolic execution tree. At each branching point both the affected logical expression and its negation are accumulated on the true and false branches, thus all of the execution paths will be linked to a unique formula over the symbolic variables.

RTEHunter detects runtime issues by traversing the symbolic execution tree and if a certain condition is fulfilled the engine reports an issue. However, the number of execution paths increases exponentially with the number of branching points thus the exploration of the whole symbolic execution tree is impossible in practice. To overcome this problem the symbolic execution engines set up different kinds of constraints over the tree. E.g. the number of symbolic states, the depth of the execution tree, or the time consumption is limited. In RTEHunter the depth of the symbolic execution tree (means symbolic state depth) and the number of states can be adjusted and the strategy of the tree traversal can also be selected.

Our goal in this work is to find the optimal parametrization of RTEHunter in terms of maximum number of states, maximum depth of the symbolic execution tree and search strategy in order to find more runtime issues in less time. The maximum depth limits the height of the tree, the maximum number of states defines its width, while the search strategy determines in which order the states in this limited size tree should be traversed.

The results show that by limiting only the depth which results a wide and flat-shaped tree the number of detected runtime errors increases exponentially in depth. However, the time consumption also increases exponentially indicating that such a flat tree configuration is not efficient in terms of the number of detected issues per time unit.

Better results can be achieved by a narrow but deeper tree. Limiting the tree by around 300 depth and 3000 states the number of detected runtime issues increased by 55% within the same time frame compared to the flat-shaped tree turning out the best configuration considering the shape of the tree.

We also developed a search strategy that improves the performance. This search strategy uses a heuristic which is based on the number of null values in a state, and directs the search towards those states where there are more null values, increasing the chance to find null pointer dereference issues. The number of the detected null pointer dereferences increased by 13% on average using this heuristic approach compared to the general breadth-fist search strategy.

References

[1] Pressman, Roger S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill Higher Education, 2001

[2] Tassey, G., *The Economic Impacts of Inadequate Infrastructure for Software Testing*, National Institute of Standards and Technology, 2002.

[3] King, James C., *Symbolic Execution and Program Testing*, Communications of the ACM, vol. 19, no. 7, pp. 385-394, 1976.