# The Optimization of a Symbolic Execution Engine for Detecting Runtime Errors

István Kádár[a]

## Abstract

In a software system, most of the runtime failures may come to light only during test execution, and this may have a very high cost.

To help address this problem, a symbolic execution engine called RTE-Hunter, which has been developed at the Department of Software Engineering at the University of Szeged, is able to detect runtime errors (such as null pointer dereference, bad array indexing, division by zero) in Java programs without actually running the program in a real-life environment.

Applying the theory of symbolic execution, RTEHunter builds a tree, called a symbolic execution tree, composed of all the possible execution paths of the program. RTEHunter detects runtime issues by traversing the symbolic execution tree and if a certain condition is fulfilled the engine reports an issue.

However, as the number of execution paths increases exponentially with the number of branching points, the exploration of the whole symbolic execution tree becomes impossible in practice. To overcome this problem, different kinds of constraints can be set up over the tree. E.g. the number of symbolic states, the depth of the execution tree, or the time consumption could be restricted.

Our goal in this study is to find the optimal parametrization of RTEHunter in terms of the maximum number of states, maximum depth of the symbolic execution tree and search strategy in order to find more runtime issues in a shorter time.

Results on three open-source Java systems demonstrate that more runtime issues can be detected in the 0 to 60 basic block-depth levels than in deeper ones within the same time frame. We also developed two novel search strategies for traversing the tree based on the number of null pointer references in the program and on linear regression that performs better than the default depth-first search strategy.

[a]University of Szeged, Department of Software Engineering Árpád tér 2. H-6720 Szeged, Hungary, E-mail: `ikadar@inf.u-szeged.hu`

# 1   Introduction

Nowadays, in software engineering it is a big challenge to produce huge, reliable and robust software systems. About 40% of the total development costs go on testing [29]; and the maintenance activities, especially the bug fixing of the system also require a considerable amount of resources [35]. In this area, symbolic execution has proven to be a practical technique for building automated test case generation and bug finding tools [6, 7, 15, 34], which supports the maintenance phase of the software engineering lifecycle.

In the context of software testing, the key goal of symbolic execution is to explore as many different program paths as possible in a given amount of time, and for each path to (1) generate a set of concrete test input values which exercises that path during a normal execution, and (2) check for the presence of various kinds of errors including uncaught exceptions, memory corruption and security vulnerabilities.

Our symbolic execution engine called RTEHunter developed at the Department of Software Engineering at the University of Szeged was developed with the goal of detecting runtime errors (such as null pointer dereference, bad array indexing, division by zero) in Java programs without actually running the program in a real-life environment.

Symbolic execution [21] is based on the notion 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, hence a symbolic state is a set of concrete program states. When the execution of the program arrives at a branching condition containing a symbolic variable which has an unknown value, the condition cannot be evaluated and the execution continues on both branches. The execution paths created this way compose a tree called the symbolic execution tree.

However, as the number of execution paths increases exponentially with the number of branching points these tools struggle to achieve scalability. To overcome this problem the symbolic execution engines set up different kinds of constraints over the tree. For example, the number of symbolic states, the depth of the execution tree, or the runtime is limited. With RTEHunter, the maximum depth of the symbolic execution tree (i.e. the symbolic state depth) and the maximum number of states can be adjusted and arbitrary strategies of the tree traversal can also be implemented.

The main aim of this study is to find the optimal parametrization of RTEHunter in terms of the maximum number of states, maximum depth of the symbolic execution tree and search strategy in order to identify more runtime issues in less time. This means we have to figure out which part of the whole execution tree contains most of the runtime issues while taking into consideration the time consumption of the exploration. Moreover, the search strategy is also essential to direct the exploration towards those states in the sub-tree where it is more likely to find issues and skip those that are supposed to be error-free. The maximum depth limits the height of the tree, and with a fixed depth the maximum number of states defines its width, while the search strategy determines the order in which the states in this

limited size tree should be traversed.

The main contributions of this study are the following:

- We discovered how the maximum number of states affects the execution time and the number of errors found without any constraint on the depth.

- We learned how the maximum number of states together with the maximum depth of the symbolic execution tree affects both the amount of runtime issues detected and the analysis time required.

- As our main contribution, we propose two novel search strategies that successfully detect more runtime issues by guiding the search towards the more error-prone parts of the source-code.

## 2  Background

### 2.1  Overview of Symbolic Execution

During execution, each program performs operations on the input data in a pre-defined order. The main idea behind symbolic execution [21] is to use symbolic variables instead of the actual data as input values and represent the values propagated during execution as symbolic expressions. A symbolic variable is a set of the possible values of a concrete variable in the program, hence a symbolic state is a set of concrete states. The output values computed by the program are then expressed as a function of the input symbolic variables.

When the execution encounters a selection control structure (e.g. an if statement) where the logical expression contains a symbolic variable whose value is unknown or uncertain, the expression cannot be evaluated, implying that the execution continues on both branches accordingly. This way all of the possible execution branches of the program can be simulated in theory.

Each state of a symbolically executed program contains a *path condition (PC)*. The path condition is a quantifier-free logical formula over the input symbolic variables with the initial value of true. It accumulates constraints which the inputs must satisfy in order to direct the execution to follow the path associated with the formula.

In addition to maintaining the path condition, symbolic execution engines make use of the so-called *constraint solver* programs. Constraint solvers are used to solve the path condition by assigning values to the symbolic variables that satisfy the logical formula at any state of the symbolic execution. Practically speaking, the solutions serve as test inputs that can be used to run the program in such a way that the concrete execution follows the execution path for which the PC was solved.
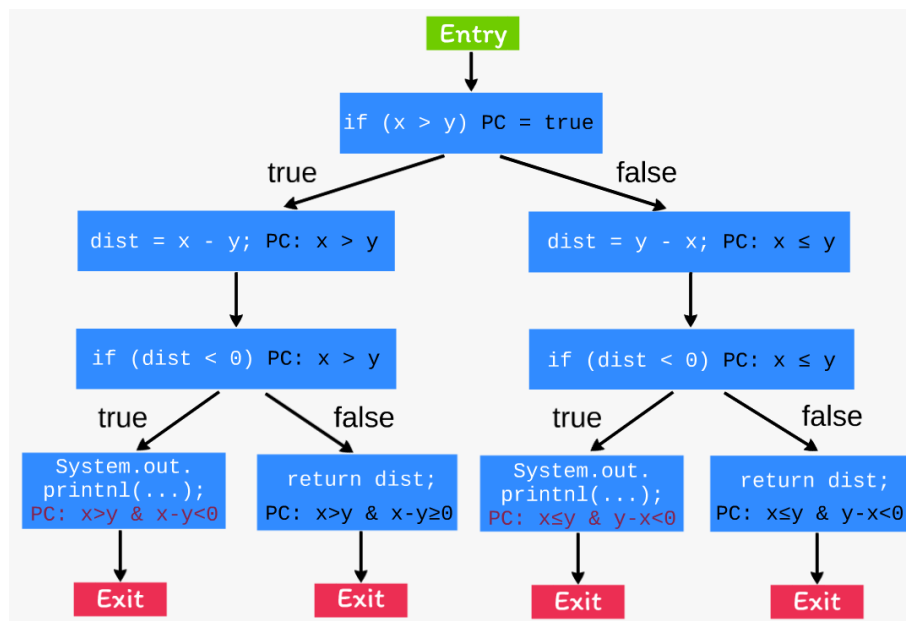
All of the possible execution paths define a connected and acyclic directed graph called the *symbolic execution tree*. Each point of the tree corresponds to a symbolic state of the program.

```
 1    int distance(int x, int y) {
 2        int dist;
 3        if (x > y) {
 4            dist = x − y;
 5        } else {
 6            dist = y − x;
 7        }
 8        if (dist < 0) {
 9            System.out.println("Error");
10        }
11        return dist;
12    }
```

(a)



(b)

Figure 1: (a) Sample code that determines the distance between two integers on the number line. (b) The symbolic execution tree of the sample code that handles variable $x$ and $y$ symbolically.

Figure 1 (a) shows a simple Java method that determines the distance between the two integer parameters x and y. The symbolic execution of this code is illustrated in Figure 1 (b) with the corresponding symbolic execution tree, showing the actual path condition. Suppose that parameters $x$ and $y$ are handled symbolically, the initial value of the path condition is true. Encountering the first if statement in line 3, there are two possibilities, namely the logical expression may be true or false; hence the execution branches and the logical expression and its negation are added to the PC.

The value of variable *dist* will be a symbolic expression: $x - y$ on the true branch and $y - x$ on the false one. As a result of the second if statement (line 8), the execution branches and the appropriate PCs are again appended. On the true branches it is obvious that the formulas are unsatisfiable (shown in red), meaning we cannot specify such $x$ and $y$ that meet the conditions. So long as the PC is unsatisfiable at a state, the sub-tree starting from that state can be pruned; then there is no sense in continuing the controversial execution.

## 2.2 RTEHunter

Now we will give a brief description of our symbolic execution engine called RTE-Hunter in order to explain the optimization approaches and investigations presented in this study.

RTEHunter was created with the goal of detecting runtime errors in Java applications without running the program in a real-life environment. In contrast with other symbolic execution tools [6, 21, 36, 37, 38], generating test cases which lead to failure is not a goal here, but rather to produce a descriptive designation of the execution path that led to a fault.

RTEHunter was developed in C++ and so far the detection of four kinds of runtime faults have been implemented: (1) null pointer dereferences, (2) array over-indexing, (3) array creation with a negative size, and (4) division-by-zero errors.

Instead of starting the symbolic execution from the *main()* method that is the entry point of a Java program, RTEHunter performs the analysis by symbolically executing each method of the system one after the other [20]. This does not mean that the engine cannot handle method calls, as the analysis is interprocedural. The engine handles method calls by placing the actual parameters onto the stack and giving the control to the callee.

The parameters of the method and the referred but not initialized variables are handled as symbols at the beginning of the symbolic execution. It is essential that we only report an error if it is proved that during the execution the value that causes the problem can be determined by constant propagation. In other words, if a method call passes a concrete null value, and RTEHunter finds a path in the called method that dereferences this parameter, we will fire an error, but if the dereferenced variable is a symbol we will not, because its value is unknown or uncertain.

The symbolic execution is performed using the language-dependent *abstract semantic graph (ASG)* [11] of the program by interpreting the nodes of this graph

in a defined order. The order is defined by the language-independent *control flow graph (CFG)* [2], which is constructed for each method. The nodes of the control flow graph are called *basic blocks*. A basic block represents a straight-line piece of code that is guaranteed to execute sequentially (i.e. it does not include any jumps or jump targets) by lining up the appropriate ASG nodes according to the sequential execution. Directed edges between any two basic blocks are used to represent jumps in the control flow. In RTEHunter, for each method analyzed the symbolic execution tree is constructed by traversing the CFG and for each basic block a symbolic state will be created in the tree. Loops and recursions are not handled in any special way and the traversal of the CFG results in simple unrolling. Figure 2 shows the control flow graph constructed for the method *distance()* shown in Figure 1 (a) and the symbolic execution tree that RTEHunter creates using the control flow graph is shown in Figure 3.

Listing 1 shows the pseudo-code of the algorithm used in RTEHunter which builds up the symbolic execution tree while executing symbolically each path with a C++-like syntax. The search-and-build strategy shown here is depth-first search. The construction of the execution tree commences with the method *search()*. Here, we first get the root state of the tree, and initialize the *strategy* object with this. Afterwards, the while loop always gets the next state to be executed. The execution of a state is performed by method called *executeState()*, which interprets the nodes that are in the basic block which the state is created from according to the semantic of the Java programing language. The strategy object provides the next state according to the implemented strategy. In this listing, the strategy is implemented in the class *DepthFirstSearchStrategy*, in which the *getNextState()* method is the essential part of the traversal. It gets the top-most element from the stack and expands this state, intending to get all of its descendants, then it puts them onto the stack. The *expandState()* method of our expander object constructs the child states according to the descendent Basic Blocks in the CFG and the information that is got from the execution of the parent states. For instance, if the parent state represents an if statement that would have two children, but the logical expression could be evaluated by execution of the parent, the *expandState()* will not provide both children, but just the appropriate one. The stack data structure (LIFO queue) provides the depth-first search traversal.

Listing 1: The main algorithm of tree building and execution in RTEHunter.

```
1   Strategy* strategy = new DepthFirstSearchStrategy(expander);
2
3   void search(StateFactory stateFactory) {
4       State *rootState = stateFactory.getRootState();
5       strategy->initialize(*rootState);
6       State* nextState = NULL;
7       while (nextState = strategy->getNextState())
8           executeState(*nextState);
9   }
10
11  class DepthFirstSearchStrategy: public SearchStrategy {
12      private:
13          std::stack<State*> stack;
14
15      public:
16          DepthFirstSearchStrategy(StateExpanderInterface& expander)
17              : SearchStrategy(expander), stack() {}
18
19          void initialize(State& rootState) {
20              stack.push(&state);
```

```
21        }
22
23        State* getNextState() {
24          State* front = stack.top();
25          stack.pop();
26          std::vector<State*> children=expander.expandState(*front);
27          for (State* child : children)
28            stack.push(child);
29          if (stack.empty())
30            return NULL;
31          return stack.top()
32        }
33      };
```
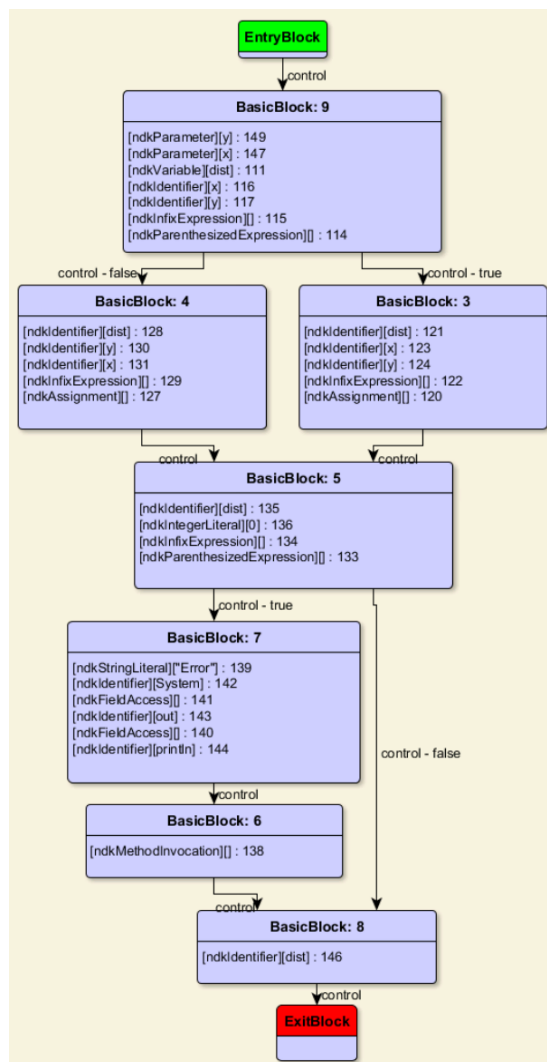
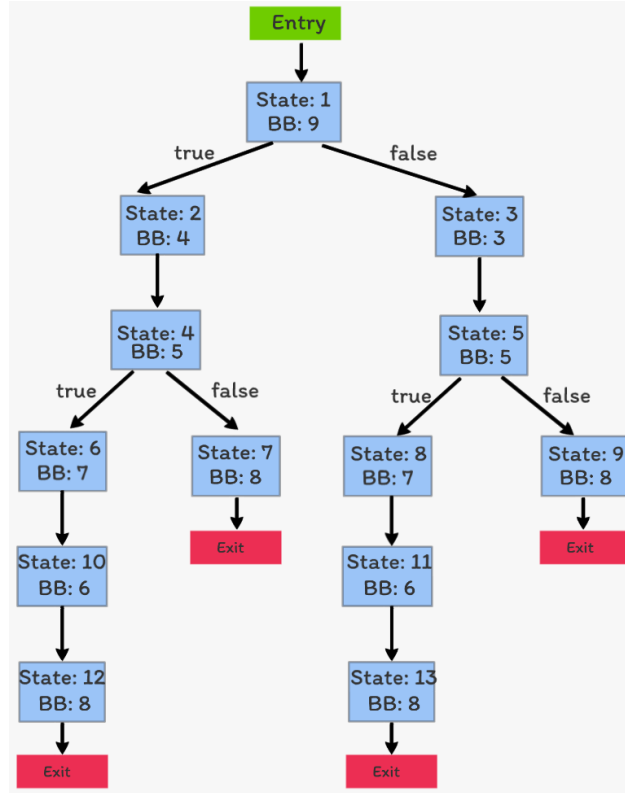Figure 2: The control flow graph (CFG) constructed for the method in Figure 1 (a).

Figure 3: The symbolic execution tree constructed by RTEHunter for the code shown in Figure 1 (a).

RTEHunter is able to handle integer, floating point, reference (including aliasing), and array type data. It models the memory by storing the variables in a special data structure optimized for memory usage. We do not store the whole variable set for each state, but only the changes. That is, if the engine wants to read a variable that was assigned in a parent state, it will be found only in the storage of the appropriate parent, but if the value a variable was changed by the current state, the updated value will be stored in the variable storage of the current state. Moreover, each state has a stack that is used for passing parameters during method invocation and for calculating expressions.

The same runtime failure can be detected repeatedly if multiple execution paths have been explored which led to the same program location. However, it is not obvious how to determine in which cases we consider two errors the same, because the original cause of two errors may be different despite the fact that the detection points are the same. For example, a reference type variable can be set to null at different locations in the program, and then dereferenced at the same place; then

this will cause a *NullPointerException*. The detection location is the same, but the causes (and the possible fixes) may differ. However, by examining the results, we realized that if multiple paths led to the same location, in most of the cases the cause is the same too because the one path is a suffix for the others. Then it seems that if multiple paths lead to the same error location, we should retain only the shorter one and filter out the others.

To limit the size of the symbolic execution trees built up for each method and the maximum depth and the maximum number of states can be specified. In accordance with the above, depth here means basic block depth. Furthermore, it is also possible to define custom search search strategies with which the traversing of the execution tree can be guided.

The output of the RTEHunter contains the errors detected that indicate their type and the execution path by a list of states from the entry point to the exact location where the error occurred. This data are written to a formatted text file, but RTEHunter has also been integrated into the SonarQube quality management platform [8].

# 3   Approach

## 3.1   Optimal Maximum Depth and State Number

As we mentioned in Section 2.2, two kinds of constraints can be set up in RTE-Hunter in order to limit the size of the symbolic execution tree and prevent a state explosion. Namely, (1) the maximum depth (which means state depth), and (2) the maximum number of states can both be adjusted. With a given maximum depth, the maximum state number defines the width of the tree. It should be added that these constraints are applied to each symbolic execution tree built for each method of the given system separately and they are not global limits for the system-wide analysis.

In our experiments, we ran RTEHunter with different depth and state number limits on three open source Java systems. The systems analyzed with their total lines of code metric are listed in Table 1.

Table 1: The chosen Java systems on which the measurements were carried out

| System | TLOC |
|---|---|
| ArgoUML | 372K |
| Jetspeed | 275K |
| JFreeChart | 329K |

In addition to the maximum depth and the maximum state number, the final shape of the symbolic execution tree is also determined by the search strategy applied.

In our experiments we sought to ascertain the optimal depth and state number; and in order to find more runtime errors in less time, we used the default depth-first search strategy.

## 3.2   Custom Search Strategies

Since the search strategy that is used to direct the traversal influences the shape of the final symbolic execution tree, it plays a significant role in finding those states where runtime errors actually occur.

In the actual stage of the traversal of the state space (i.e. the symbolic execution tree), a search strategy tells us from which state among the current leaf states the exploration has to continue, i.e. which state has to be expanded as the next step of the traversal.

In contrast to depth-first search where the actual leaf states placed into a LIFO (last in, first out) queue, in our custom search strategies a score is assigned to every current leaf, which will be placed into a priority queue. When the engine have to chose a state as the next step, it chooses the one with the highest score to continue the traversal. The implementations of these strategies are very similar to the code of class *DepthFirstStrategy* shown on Listing 1, but the stack member is replaced by a priority queue that orders the states by score, and the top is the one has the highest score.

Next, we will describe two new search strategies which implement heuristics to direct the search towards the potential runtime issues.

### 3.2.1   The Null-heuristic Search Strategy

This search strategy attempts to drive the traversal to find more null pointer dereference issues. Our motivation of focusing on null pointer dereferences is that according to static analysis the most common checks against exceptions are null-checks in Java sources, implying that this is the most common runtime issue that may occur [12, 32]. We also discovered that this type of runtime error is the most common one that RTEHunter encounters.

For each state we summarize the number of reachable reference-type values (variable values, literals, function return values, etc.), whose value is *null* at the current symbolic state of the given Java program. To continue the traversal, the engine chooses the state with the highest number of null values assigning higher probability value to find possible null pointer dereferences in that state and in the sub-tree obtained from it.

### 3.2.2   A Linear Regression-Based Search Strategy

We developed a search strategy that supports the detection of not just null pointer dereferences, but also all the four types of issues that RTEHunter is able to detect. To implement such a search strategy we used a linear regression model that assigns a score to each leaf state during the search. The score is the estimated number of

runtime issues that might have been detected in the sub-tree reachable from the state. We chose linear regression because it can be applied on continuous class labels and it provides a relatively quick result for an unseen example.

The training data of the linear regression model contains one training example for each state got from symbolic execution trees that were traversed previously by the engine. The label (i.e. the supervisory signal) for each example is the number of runtime issues that were detected in the sub-tree under the state that the example belongs to.

We defined five attributes as predictors that can be determined for each state. Attributes may contain both static source-code information and dynamic information that the symbolic execution supplies.

The attributes are the following:

1. *The depth of the state in the symbolic execution tree.* If there is a tendency of how deep the significant part of the faults occur, the information will be encoded into the model.

2. *The number of null values in the state*, as described in Section 3.2.1.

3. *The sum of the number of zero numeric type values (variable values, literals, function return values, etc.) in the state, and the number of division operators reachable from the state according to the control flow graph in 15 basic block depth.* Here, we combined the dynamic information of zero values and the static information of the number of division operators in the possible future of the execution. This attribute is a heuristic for finding division-by-zero errors.

4. *The Logical Lines of Code (LLOC) metric of the method that the state belongs to.*

5. *The cyclomatic complexity metric [26] of the method that the state belongs to.*

As lines of code (LOC) and cyclomatic complexity have proved to be promising defect predictors [27, 28], attributes 4 and 5 should be useful in our heuristic. Both of them were calculated using static a source code analyzer tool called *SourceMeter*[1].

We applied the linear regression algorithm implemented in the *Shark* machine learning library [18].

## 4 Results

### 4.1 Optimal Maximum Depth and State Number

In order to ascertain the optimal limitations of the symbolic execution tree built by RTEHunter with the goal of finding runtime issues in a minimal time frame, we performed numerous analyses and applied different constraints.
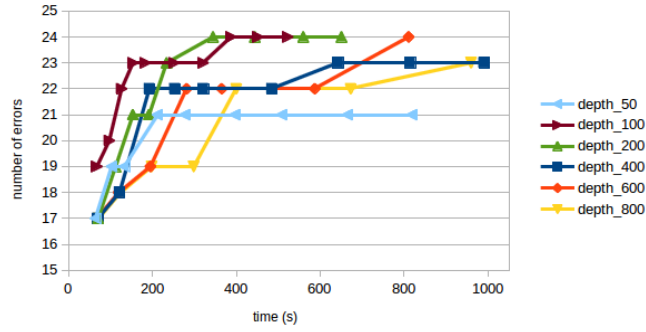
---

[1]http://www.sourcemeter.com/

Figure 4: The increase in the number of errors at analysis time using different depth limits in ArgoUML.
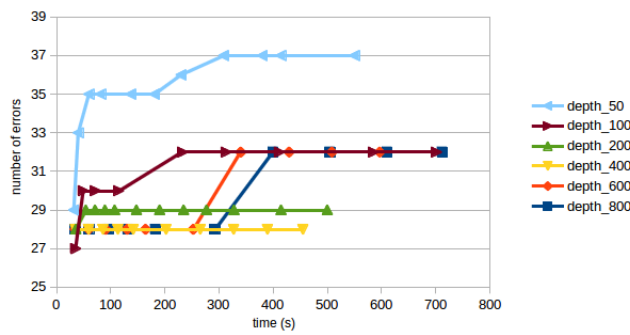


Figure 5: The increase in the number of errors at analysis time using different depth limits in Jetspeed.
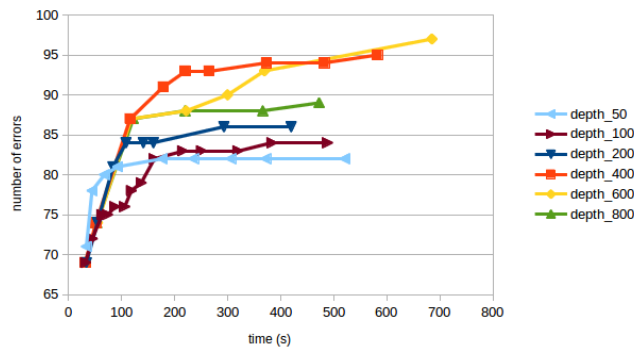


Figure 6: The increase in the number of errors at analysis time using different depth limits in JFreeChart.
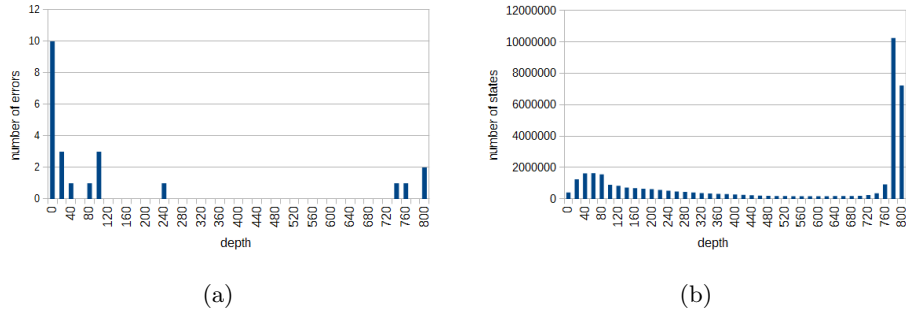
(a)                                    (b)

Figure 7: (a) The distribution of the errors at different depth levels in ArgoUML. (b) The distribution of the states at different depth levels in ArgoUML.

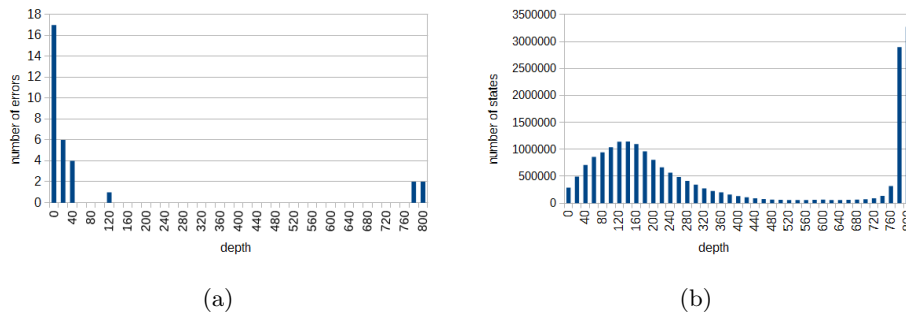

(a)                                    (b)

Figure 8: (a) The distribution of the errors at different depth levels in Jetspeed. (b) The distribution of the states at different depth levels in Jetspeed.
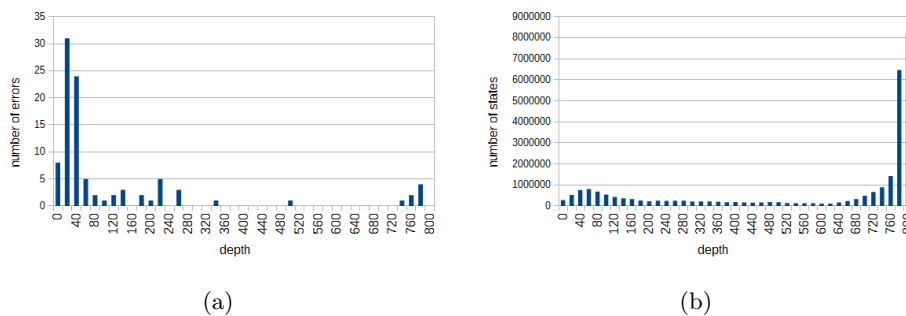


(a)                                    (b)

Figure 9: (a) The distribution of the errors at different depth levels in JFreeChart. (b) The distribution of the states at different depth levels in JFreeChart.

First of all, we found that the number of executed states is closely correlated with the analysis time. The Pearson-correlation coefficients are all above 0.99, which is a strong positive correlation, and it means that the high state limit corresponds to a high run-time. The results are significant at $p < 0.05$. However, the correlation coefficients among the number of states and the number of issues found range from 0.3 to 0.8, indicating a weaker relationship, and the results are not significant in many cases at $p < 0.05$. Hence the number of states seems to determine the execution time, but the number of errors probably depends on other factors as well.

To understand the role of maximum depth, we ran RTEHunter with different depth limits, at each depth limit level and with different maximum state sizes, and this allowed us to investigate how the results varied by increasing the analysis time. The depth limits chosen were 50, 100, 200, 400, 600 and 800, for each the maximum state values differ from 200 to 10,000. The results were put onto line diagrams shown in figures 4, 5 and 6 for ArgoUML, Jetspeed and JFreeChart, respectively. The diagrams show how the number of errors grows in time at each depth limit level. On each line the dots represents an RTEHunter run with a specific state number limit, which are formed to extend the analysis at least nearly 500 seconds.

In general, each line starts increasing, and after a while the number of errors does not increase anymore. The reason why the number of errors stagnate after a time may be because all of the errors were found at that depth level, and to detect new ones the analysis needs to go deeper. The depth limit is considered to be better which rises rapidly and with which the engine detects higher number of errors. The best depth limit varies from system to system.

As regards ArgoUML, the 100-depth configuration increases the most rapidly, but at the depth of 200 one can find 24 errors in slightly shorter time. By decreasing the depth limit to 50, the results get worse, and the 400- 600- and 800-depth configurations also perform worse.

With Jetspeed, the 50-depth limit is far better than the others. While with the depths of 100, 600 and 800 we can reach only 32 errors under 500 seconds, we managed to detect 37 errors with the 50-depth configuration. These results suggest that the majority of the errors in Jetspeed are at a depth of below 50.

Among the investigated depth limits, the 400-depth may be considered to be the optimum in the case of JFreeChart. However, the 50-depth one starts to rise the most rapidly, then it stops growing after 100 seconds. After 600 seconds the 600-depth limit becomes slightly better than the 400-depth one, but in general the 400-depth one performs the best.

To understand why the above-mentioned depth-limits performed the best in the experiments, we analyzed the distribution of the errors at different depth levels. Subfigures (a) of figures 7, 8 and 9 show that the number of errors found at each depth level. The depth-level intervals formed to be 20-length intervals in each diagram. The height of the bars in each diagram represents the number of errors found in 20-length depth intervals. The data is derived from an 800 depth and 15,000 state limit run, with which we attempted to analyze as big symbolic

execution trees as the memory consumption made possible.

In general, the same pattern appears to be present in all three systems. The number of errors are significant at shallower levels, then in the middle just a few of them were detected, and close to the depth limit errors occur again but not at such high numbers as in the shallower levels. E.g. the error distribution of Jetspeed in Figure 8 (a) tells us that the majority of the errors were found at a depth limit of 40 or less, which explains why the 50-depth limit is so satisfactory in Figure 5. Although deeper configurations reached more states, the error density is rather low there, hence we wasted time spent in executing these.

We have also plotted the distributions of all the states that were explored by RTEHunter in figures 7, 8 and 9 (b). These diagrams show the overall shape of the symbolic execution trees traversed through the analysis. A similar pattern can be seen in the error distribution diagrams, which partially explains the error distribution: at depth levels where more states are explored, more errors can be found. However, there are many more states near the 800-depth maximum-limit than in the shallow parts of the tree, but the number of errors is higher in the shallow levels than at the bottom. This leads us conclude that the the runtime issues that RTEHunter can detect are in general more common between levels 0 to 60 basic block depths compared to the deeper levels.

It should be added that the search strategy which is used to explore the state space has a marked effect on the state distribution, and hence on the error distribution too.

## 4.2  Null-heuristic Search Strategy

The goal of the null-heuristic search strategy is to increase the number of runtime issues detected in the given time frame, compared to the default depth-first search. In particular, we focused on the number of null pointer dereferences. To make a comparison, we used those configurations which were found to be the best in Section 4.1 as a reference. The maximum depth for ArgoUML is 100, 50 for Jetseed and 400 for JFreeChart. We also chose the same state number limit sequence to expand the analysis time as before. With these parameters, but this time with our novel null-heuristic we repeated the experiments. The results of these experiments are shown in Figure 10.

In each case the null-heuristic approach performed better, because the number of detected issues increases more rapidly and the values higher, i.e. it found more runtime issues in less time which surely confirms the efficiency of our algorithm. It is worth mentioning here that over 90% of the errors found in these systems are null pointer dereferences, and the new search strategy applied does not modify this ratio significantly.

What is more interesting is that the same analysis sequence with the null-heuristic approach finished faster than before. E.g. in the case of ArgoUML the last analysis (the last dot on the line) lasted 431 seconds with the null-heuristic, and 522 seconds with the conventional DFS. This point is surprising because we need to calculate the number of null reference values for each state and also maintain
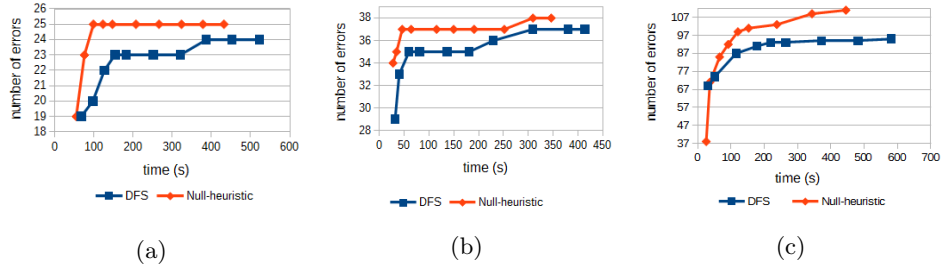
Figure 10: The efficiency of null-heuristic search compared to the default depth-first search on ArgoUML (a), Jetspeed (b), JFreeChart (c).

a priority queue to keep the leaves in for the null-heuristic algorithm. Probably the reason why it is still faster is that it guides the search towards states whose execution time is shorter.

## 4.3   Linear Regression Based Search Strategy

Table 2: The number of detected runtime issues with the linear regression based search strategy (LR based) compared to the default depth-first search (DFS)

|  |  | 500 max state | 1000 max state | 1500 max state |
|---|---|---|---|---|
| ArgoUML | DFS | 22 | 23 | 23 |
| (max depth: 100) | LR based | 51 | 54 | 55 |
| Jetspeed | DFS | 33 | 35 | 35 |
| (max depth: 50) | LR based | 66 | 74 | 73 |
| JFreeChart | DFS | 91 | 93 | 94 |
| (max depth: 400) | LR based | 122 | 131 | 138 |

In the evaluation of the linear regression-based search strategy, we use 10-fold cross-validation on each system in the following way. Firstly, to form the folds we sort the methods of the system by lines of code (LOC). In the sorted list, each $j^{th} method$ is placed into $fold_i$ if equation $j\ mod\ 10\ =\ i$ is satisfied. In other words, every tenth method will go to the same fold (see Figure 11), ensuring that no fold differs too much from the others in the length of the methods contained.

After running RTEHunter on 10 folds, we summarized the number of errors that were detected in each fold. The structure of the folds ensures that each error is counted only once. The number of errors found using this strategy (LR-based) is shown in Table 2 and it is compared to the errors found by the default depth-first search strategy (DFS). With all the subject systems we examined the depth limit that was found to be the best in Section 4.1 using DFS: 100 for ArgoUML, 400 for JFreeChart, and 50 for Jetspeed. As regards the maximum state number constraint, we provide results for 500, 1000 and 1500 maximum state values.
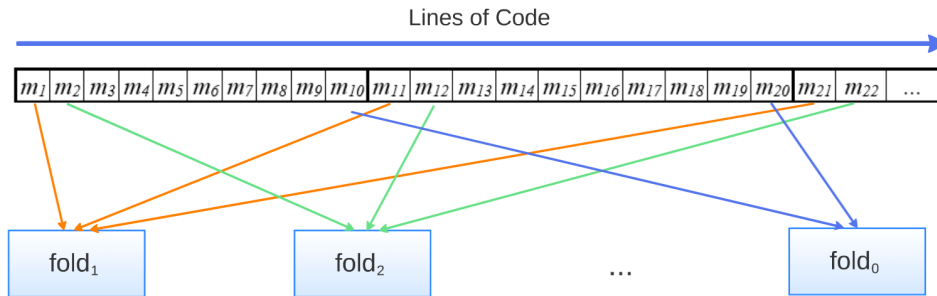
Figure 11: The formation of the folds used to perform 10-fold cross validation.

As the results demonstrate, our novel algorithm outperforms the default one in each case. In ArgoUML and Jetspeed, we found more than twice as many errors as we did with DFS. However, in the case of JFreeChart this ratio is smaller, the difference being still significant: it formed around 30 to 40 more issues were discovered.

The reason why we do not present the runtime here is due to the implementation details of the 10-fold cross validation. Currently, before RTEHunter starts the analysis of each fold, it has to load the ASG and then rebuild the CFG for architectural reasons. This introduces an overhead, which is not present in the case of a conventional run. Apart from this shortcoming, the difference in the number of detected issues is still significant.

# 5 Threats to Validity

In the present study we do not focus on the precision of RTEHunter. Some of the issues found may be false positives and these may change or invalidate the result of our investigations. Manual validation needs a considerable amount of work in the case of such a high number of errors presented here. However, in a future work we plan to perform the manual validation and repeat the experiments with the updated dataset. We also intend to examine how the false positive rate is affected by the traversal strategy because we wish to avoid situations where a deeper analysis on a path produces more false positives for instance because of an incorrectly handled coding pattern.

# 6 Related Work

**Symbolic execution engines and similar tools.** The idea of symbolic execution is not new, and the first publications and execution engines appeared in the 1970's. One of the earliest papers is by King, which laid down the fundamentals of symbolic execution [21] and presented the EFFIGY system that is able to execute PL/I programs symbolically. Even though EFFIGY handles only integers

symbolically, it is an interactive system with which the user is able to examine the process of symbolic execution by placing breakpoints and saving and restoring states. Another paper from the 1970's is by Boyer et al., who presented a similar system called SELECT [3] that can be used for executing LISP programs symbolically. The users are allowed to define conditions for variables and return values and get back whether these conditions are satisfied or not as an output. The system can be applied for test input generation and in addition, for each path it returns the path condition over the symbolic variables.

Starting from the last decade interest in this technique has been steadily growing, and numerous programs have been developed that seek to dynamically test input generation using symbolic execution. There are also approaches and tools for generating test suites for .NET programs using symbolic execution. Pex [36] is a tool that automatically produces a small test suite with high code coverage for .NET programs using dynamic symbolic execution, similar to path-bounded model-checking. Jamrozik et al. introduce an extension of the previous approach called augmented dynamic symbolic execution [19], which seeks to produce representative test sets with DSE by augmenting path conditions with additional conditions that enforce target criteria such as boundary or mutation adequacy, or logical coverage criteria. Experiments with the Apex prototype demonstrate that the resulting test cases can detect up to 30% more seeded defects than those produced with Pex.

In Section 2.2, we mentioned that the results of RTEHunter can be published into a SonarQube instance [8]. SonarQube has its own symbolic execution engine for Java language, which similar to us is intended to find tricky bugs that are almost undetectable by developers unaided. It is able to find three kinds of issues compared to our five: (1) null pointer dereference, (2) unclosed resource and (3) a rule named "condition should not unconditionally evaluate to true or false". Based on our thorough testing of the tool it has a constraints solver, but it does not handle function calls, meaning it might miss serious errors like null pointer dereferences that RTEHunter is able to detect even if the control flows through multiple function calls. Another difference with RTEHunter is that if an issue is found, SonarQube presents only that part in the source code where the issue actually appears, and it does not give the path that leads to the fault which can really help the developer in the debugging work. It should be added that widely used static analysis tools like PMD [1] and FindBugs [17] can also be integrated into SonarQube to find issues and bugs. PMD performs syntactic checks on the source-code by building an ASG first and it does not carry out any deeper analysis to detect runtime issues like a symbolic execution tool. Beyond syntactic checks, FindBugs syntactically matches source code to known suspicious programming practice and also uses data-flow analysis to check for bugs like null pointer dereference, class cast exception and array index out of bounds exception in an intraprocedural way, but this approach does not present the results as an interprocedural symbolic execution.

**The handling of the state explosion in symbolic execution.**    To reduce the state space of the symbolic execution, Symbolic PathFinder [30] based on the Java PathFinder model checker offers a number of options. Similar to RTEHunter, the

maximum depth of the symbolic execution tree can be specified, and the number of elementary formulas in the path condition can be restricted. Another possibility is that with options we can restrict the value ranges of the integer and floating point type symbolic variables. In addition, Symbolic PathFinder lazily initializes object references and uses types to infer aliasing.

Bucur et al. [4] addresses the problem of path explosion by parallelizing symbolic execution in such a way that it scales well on large clusters of cheap commodity hardware. The system, called Cloud9, can automatically test real systems that interact in complex ways with their environment.

Another approach for reducing the state space, presented by Chipounov [9], is not to execute the whole program symbolically, but just portions of it. The engine can start the symbolic execution at arbitrary places of a whole system, including applications, libraries, the operating system, and device drivers. It seamlessly goes back and forth between symbolic and concrete execution, while transparently converting system states from symbolic to concrete and back. A similar approach is used for testing NASA Software [31]. The idea is also to start the program in normal mode as in a real-life environment, then at given points (e.g. at more complex or problematic parts in the program) it can switch to the symbolic execution mode. The CUTE and jCUTE systems [33] constructed by Sen and Agha, are also concolic executors, where they start at an arbitrary function by initializing pointers based at first on a simple heap with abstract addresses and incrementally increase the heap complexity in subsequent runs.

In order to make symbolic execution more scalable, Majumdar and Xu propose using symbolic grammars to guide symbolic execution by reducing the space of possible inputs [25]. Godefroid et al. utilized a similar approach. They set up the grammar-based specification of highly-structured inputs of symbolic execution such as compilers and interpreters [14].

Another approach is to reuse and merge the paths that were explored earlier. Also, reusing the analysis of lower-level functions in subsequent computations improves the scalability of symbolic execution [13].

Loops and recursions with conditions that cannot be evaluated during the symbolic execution result in infinite constructs that can explode the state space without benefit. For this reason, the handling of these constructs may have significant effects. CBMC is a Bounded Model Checker for C and C++ programs [10]. CBMC is able to verify array bounds, exception handling correctness, pointer safety and user defined assertions as well. In bounded model checking, the potentially infinite constructs (e.g. while loops, recursion) are unwound only $n$ times, where this number $n$ is the upper bound. CBMC pre-processes the program into an equivalent program that uses only while, if, goto statements, and assignments. Next, all while loops are unwound using the following transformation $n$ times:

*while (cond) instruction;* $\rightarrow$ *if (e) {instruction; while(cond) instruction}*

The last while loop is replaced by assertion *!cond*, which ensures that the program never performs more iterations. This unwinding assertion is verified along with the user defined assertions, and if it fails, one more iteration is required in the unwinding. Afterwards, the program only consists of if instructions, assignments,

assertions, labels, and forward goto instructions, and it is then transformed into static single assignment (SSA) form, from which a bit vector equation is created together with the target rule to be verified. If this equation is satisfiable, the tool finds a violation. The mechanism presented here of unrolling loops only a certain number of times would be a good idea to integrate it into RTEHunter to reduce the state space generated by loops. Currently, we use a simple unrolling until we reach the depth or the overall state limit. Another strategy would be to recognize patterns in the basic block sequence during a loop unwinding. For example, when a basic block is visited too many times, the execution is deep inside a recursion or a loop. This strategy works well for the Clang Static Analyzer [22].

**Search heuristics in symbolic execution.**    One of the key mechanisms used by symbolic execution tools to prioritize path exploration is the use of search heuristics. Most heuristics focus on achieving high statement or branch coverage, but they could also be employed to optimize other desired criteria. The main difference compared to our study is that we optimize for execution time and also for the number of detected issues.

KLEE [6] is another symbolic execution engine that seeks to automatically generate tests that achieve high coverage. Similar to our approach, it is possible to use various heuristics to prioritize the most interesting paths first. KLEE selects the next state to run by interleaving the two search heuristics, namely random state selection and a coverage-optimized search that attempts to select states likely to cover new code. Currently, our search heuristics do not incorporate any test coverage information, but in the future we intend to examine this possibility. Random exploration proved to be an efficient test generation approach by Burnim et al [5] as well.

The EXE (EXecution generated Executions) [7] presented by Cadar et al. at Stanford University is an error checking tool designed for generating input data on which the program terminates with failure. The heuristic in EXE favors previously visited statements that were run the least number of times.

The AUSTIN tool applies fitness functions to drive an evolutionary search of the test input space with dynamic symbolic execution [23].

Ma et al. [24] focus on debugging scenarios when the developer already knows about the faulty line, but they might not know exactly how to reproduce the failure or even whether it is reproducible. The approach also applies search strategies that try to direct the symbolic execution to the target line. One strategy is the *shortest-distance symbolic execution (SDSE)*, where we pick the path that currently has the shortest distance to the target line according to the control flow graph (CFG) of the program. The other one starts at the target line and works backward until it finds a realizable path from the start of the program, using standard forward symbolic execution as a subroutine.

In general, our problem lies in the domain of Search-Based Software Engineering (SBSE), where search-based optimization algorithms are used to address problems in software engineering - e.g. to figure out the smallest set of test cases that cover all branches in this program or the set of requirements that balances software

development cost and customer satisfaction. Harman et al. give a comprehensive survey addressing this area [16]. In our case, we look for the symbolic execution tree that has smallest exploration time and covers the greatest number of runtime issues.

# 7 Conclusions and Future Work

The main goal of this study was to optimize the RTEHunter symbolic execution engine to detect more runtime issues in less time. Because of the path explosion problem, the limitation of the state space of the symbolic execution assumes a major importance in this scenario. In the empirical investigations on three open-source Java systems, it turned out that adjusting the maximum number of states for the symbolic execution trees has an effect on the execution time, but not on the number of issues found. However, the constraints on the depth of the tree is more important in the detection of runtime errors. We found different optimal depth limits for the three different systems, but we can say that errors occur more often at the basic block depth of 0 to 60 compared to the deeper levels in the systems we analyzed, but it also strongly depends on the search strategy that is applied.

Here, we propose two novel search strategies that seek to guide the symbolic execution towards the more error prone source-code fragments using both static and dynamic information. The null-heuristuc search strategy performs better by finding up to 16 % more errors within the same time frame than those found using the default depth-first search. The linear regression-based heuristic also outperforms DFS, and it can detect over twice as many errors in ArgoUML and Jetspeed.

In the future, we would like to include more systems into the empirical experiments and this may help us to find the optimal depth and state number limits of symbolic execution tree in general, which should make RTEHunter and other symbolic execution engines more efficient. We also plan to develop more efficient search strategies, by adding new features and applying other machine learning approaches. Also, we think that the investigation of including test coverage information in the search heuristic might be a promising approach. We would also like to manually validate the errors reported by the RTEHunter to discover its precision in practice.

In this study we did not place any emphasis on the practical usage of the search strategies developed in a real-life product or examine which strategy is good for which type of error. The null-heuristic search might be good for finding null dereferences, but it might also degrade the quality of detecting other types of issue. In real-life applications, one option might be to develop specific, well performing search strategies for each issue type. However, it may be time consuming to rerun the symbolic execution for each type. Another option might be to develop one complex search strategy with general predictors that performs well for all or for most of the issues. This approach might require less time because we need to construct the state space only once, but it might not scale well as the number of checks increase or do not perform as well as the issue-specific heuristics. We plan to investigate this in more detail as well.

# 8    Acknowledgment

# References

[1] Pmd/java. `https://pmd.github.io/`. Accessed: 2017-03-21.

[2] Allen, Frances E. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[3] Boyer, Robert S., Elspas, Bernard, and Levitt, Karl N. SELECT – a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM.

[4] Bucur, Stefan, Ureche, Vlad, Zamfir, Cristian, and Candea, George. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 183–198, New York, NY, USA, 2011. ACM.

[5] Burnim, J. and Sen, K. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 443–446, Washington, DC, USA, 2008. IEEE Computer Society.

[6] Cadar, Cristian, Dunbar, Daniel, Engler, Dawson R, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[7] Cadar, Cristian, Ganesh, Vijay, Pawlowski, Peter M., Dill, David L., and Engler, Dawson R. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM.

[8] Campbell, G. Ann and Papapetrou, Patroklos P. *SonarQube in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2013.

[9] Chipounov, Vitaly, Georgescu, Vlad, Zamfir, Cristian, and Candea, George. Selective Symbolic Execution. In *5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[10] Clarke, Edmund, Kroening, Daniel, and Yorav, Karen. Behavioral consistency of c and verilog programs using bounded model checking. In *Proceedings of the 40th annual Design Automation Conference*, pages 368–371. ACM, 2003.

[11] Ferenc, R., Beszédes, Á., Tarkiainen, M., and Gyimóthy, T. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM'02)*, pages 172–181. IEEE Computer Society, IEEE Computer Society, oct 2002.

[12] Flanagan, Cormac and Leino, K. Rustan M. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517, London, UK, UK, 2001. Springer-Verlag.

[13] Godefroid, Patrice. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM.

[14] Godefroid, Patrice, Kiezun, Adam, and Levin, Michael Y. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, June 2008.

[15] Godefroid, Patrice, Klarlund, Nils, and Sen, Koushik. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[16] Harman, Mark, Mansouri, S Afshin, and Zhang, Yuanyuan. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. *Department of Computer Science, King's College London, Tech. Rep. TR-09-03*, 2009.

[17] Hovemeyer, David and Pugh, William. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[18] Igel, Christian, Heidrich-Meisner, Verena, and Glasmachers, Tobias. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008.

[19] Jamrozik, Konrad, Fraser, Gordon, Tillman, Nikolai, and Halleux, Jonathan. Generating Test Suites with Augmented Dynamic Symbolic Execution. In *Tests and Proofs*, volume 7942 of *Lecture Notes in Computer Science*, pages 152–167. Springer Berlin Heidelberg, 2013.

[20] Kádár, István, Hegedűs, Péter, and Ferenc, Rudolf. Runtime exception detection in java programs using symbolic execution. *Acta Cybernetica*, 21(3):331–352, 2014.

[21] King, James C. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[22] Kremenek, Ted. Finding software bugs with the clang static analyzer. *Apple Inc*, 2008.

[23] Lakhotia, Kiran, McMinn, Phil, and Harman, Mark. An empirical investigation into branch coverage for c programs using {CUTE} and {AUSTIN}. *Journal of Systems and Software*, 83(12):2379 – 2391, 2010.

[24] Ma, Kin-Keung, Phang, Khoo Yit, Foster, Jeffrey S., and Hicks, Michael. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 95–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] Majumdar, Rupak and Xu, Ru-Gang. Directed test generation using symbolic grammars. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 134–143, New York, NY, USA, 2007. ACM.

[26] McCabe, Thomas J. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[27] Menzies, Tim, Greenwald, Jeremy, and Frank, Art. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[28] Moser, Raimund, Pedrycz, Witold, and Succi, Giancarlo. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, New York, NY, USA, 2008. ACM.

[29] Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, November 2001.

[30] Păsăreanu, Corina S. and Rungta, Neha. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[31] Păsăreanu, Corina S., Mehlitz, Peter C., Bushnell, David H., Gundy-Burlet, Karen, Lowry, Michael, Person, Suzette, and Pape, Mark. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[32] Ryder, Barbara G., Smith, Donald, Kremer, Ulrich, Gordon, Michael, and Shah, Nirav. A Static Study of Java Exceptions Using JESP. In *Proceedings of the Ninth International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 67–81. Springer-Verlag, 2000.

[33] Sen, Koushik and Agha, Gul. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Berlin, 2006. Springer-Verlag.

[34] Song, JaeSeung, Ma, Tiejun, Cadar, Cristian, and Pietzuch, Peter. Rule-Based Verification of Network Protocol Implementations Using Symbolic Execution. In *Proceedings of the 20th IEEE International Conference on Computer Communications and Networks (ICCCN'11)*, pages 1–8, 2011.

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

[36] Tillmann, Nikolai and De Halleux, Jonathan. Pex: White Box Test Generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[37] Visser, Willem, Păsăreanu, Corina S., and Khurshid, Sarfraz. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[38] Xie, Tao, Marinov, Darko, Schulte, Wolfram, and Notkin, David. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 365–381, Berlin, Heidelberg, 2005. Springer-Verlag.