SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2020;**30**:e1719.
Published online 21 November 2019 in Wiley Online Library (wileyonlinelibrary.com). https://doi.org/10.1002/stvr.1719

SPECIAL ISSUE PAPER

Facilitating program performance profiling via evolutionary symbolic execution

Andrea Aquino[1], Pietro Braione[2], Giovanni Denaro[2,*,†] and Pasquale Salza[3,*,†]

[1]*USI Università della Svizzera italiana, Switzerland*
[2]*University of Milano-Bicocca, Italy*
[3]*University of Zurich, Switzerland*

SUMMARY

Performance profiling can benefit from test cases that hit high-cost executions of programs. In this paper, we investigate the problem of automatically generating test cases that trigger the worst-case execution of programs and propose a novel technique that solves this problem with an unprecedented combination of symbolic execution and evolutionary algorithms. Our technique, which we refer to as 'Evolutionary Symbolic Execution', embraces the execution cost of the program paths as the fitness function to pursue the worst execution. It defines an original set of evolutionary operators, based on symbolic execution, which suitably sample the possible program paths to make the search process effective. Specifically, our technique defines a memetic algorithm that (i) incrementally evolves by steering symbolic execution to traverse new program paths that comply with execution conditions combined and refined from the currently collected worse program paths and (ii) periodically applies local optimizations to the execution conditions of the worst currently identified program path to further speed up the identification of the worst path. We report on a set of initial experiments indicating that our technique succeeds in generating good worst-case test cases for programs with which existing approaches cannot cope. Also, we show that, as far as the problem of generating worst-case test cases is concerned, the distinguishing evolutionary operators based on symbolic execution that we define in this paper are more effective than traditional operators that directly manipulate the program inputs.

Received 25 March 2019; Revised 25 September 2019; Accepted 2 October 2019

KEY WORDS: genetic algorithms; software engineering; symbolic execution; worst-case execution time

1. INTRODUCTION

Performance profiling is a fundamental task in many practical settings of software development, to identify, debug and fix performance bottlenecks of software programs. A core activity is executing the target program with a profiler, a tool that monitors the execution and provides detailed figures on the extent to which various parts of the implementation at different granularity levels, for example, components, functions, source and binary instructions, impact the execution costs. Typical information to be collected include the partitioning of the execution time across different program functions, or the frequency with which specific steps of an algorithm execute, or the impact on the

*Correspondence to: Giovanni Denaro, Università degli Studi di Milano-Bicocca Scuola di Scienze.
Pasquale Salza, University of Zurich, Switzerland.
†E-mail: denaro@disco.unimib.it; salza@ifi.uzh.ch

use of critical resources, for example, memory consumption or yet hints of security vulnerabilities with respect to inputs that an adversary can exploit for denial of service attacks. A critical asset for achieving successful performance profiling are the test cases that developers use to conduct the profiling.

In this paper, we investigate a novel approach to solve the problem of automatically generating test cases for performance profiling. In particular, we look into the particular instance of this problem that consists of automatically generating a test case that triggers the Worst-Case Execution Time (WCET) of a program, a problem that we call 'WCET testing'.

Traditional techniques for WCET analysis of hard real-time systems include steps based on symbolic execution to prune infeasible execution paths, but they do not generally address the generation of WCET test cases [1–7]. These techniques exploit symbolic execution on restricted parts of the overall system, to tune higher lever static analysis procedures.

More recently, the techniques Worst-case Inputs from Symbolic Execution (WISE) [8] and Symbolic Path Finder Worst-Case Analysis (SPF-WCA) [9] introduced techniques for WCET testing based on symbolic execution. The core idea of WISE and SPF-WCA is to infer the structure of the worst program path by accomplishing the 'exhaustive symbolic execution' of the target program for a simplified version of the problem, which they obtain by constraining the program inputs to a user-defined small bound, thus controlling for the amount of paths that must be analyzed. Then they extrapolate the information of the worst program path observed under the assumed small bound, to synthesize a path selection heuristic, which they call a 'guidance policy', and use the guidance policy to steer the symbolic execution of the program in the target (unconstrained) scope. If successful, the guidance policy allows them to identify the worst-case path by visiting a small subset of the possible program paths.

The new approach that we present in this paper is motivated by the observation of two crucial limitations of the approach embraced by the techniques WISE and SPF-WCA. First, the possibility of synthesizing successful guidance policies depends on the existence some generalizable structural regularity in the series of worst program paths that correspond to the target program for increasing inputs. Unfortunately, the existence of such regularity cannot be taken for granted. Second, some programs simply have by-design radically different behaviours for small and large inputs, respectively. When these issues occur for a program under test, both the techniques WISE and SPF-WCA result in either an ineffective guidance policy that selects an unmanageable amount of paths or even a misleading policy that may lead to reporting a wrong WCET test case. In Section 2, we show example programs that provide compelling evidence of these issues. Moving froward from these motivating observations, our approach to WCET testing defines an unprecedented combination of symbolic execution [10–12] and evolutionary algorithms [13, 14], which we call Evolutionary Symbolic Execution (ESE) that:

1. relies on symbolic execution to analyze a subset of the control-flow paths of the program under test and identify the execution conditions of the program paths in the form of *path condition* formulas over the symbolically represented program inputs;
2. searches for the worst feasible program path with an evolutionary algorithm based on a cost function, namely, the amount of executed instructions, which it measures against each program path during symbolic execution, and uses as the fitness function to steer the evolutionary algorithm to probabilistically select increasingly worse program paths;
3. solves the path condition of the worst identified program path to a satisfying assignment of the program inputs, thus producing a test case that concretely executes the worst program path.

As main distinctive characteristic with respect to WISE and SPF-WCA, our ESE approach analyses the program directly in the target (large) scope, thus avoiding both the burden for the testers of having to identify any suitable way of scaling from small to large inputs and the issues due to a possible mismatch between the behaviour of the program with small and large inputs, respectively. ESE embraces a meta-heuristic path selection strategy based on a 'memetic' algorithm. Memetic algo-

DOI: 10.1002/stvr

rithms are evolutionary algorithms that hybridize genetic and local search algorithms, such that the candidate solutions identified with a genetic algorithm can be improved with focused optimizations within the local search algorithm. The ESE memetic heuristic is driven by the WCET fitness function that measures the execution cost of the program paths as the number of executed instructions and, in turn, includes (i) a genetic algorithm that fosters the combination of the execution conditions from the already explored program paths, aiming to reveal sets of execution conditions that correspond to incrementally worse program paths and (ii) a local search procedure, executed at regular intervals, that attempts atomic changes to the current path condition, aiming to further refine the decision sequences of the worst program path identified so far.

A core novelty of ESE with respect to other search-based test generation algorithms is concerned with the evolutionary operators of the memetic algorithm, which ESE defines based on computing and manipulating symbolic path conditions, whereas traditional dynamic search-based testing algorithms commonly rely on evolutionary operators that directly manipulate the concrete program inputs of the incrementally generated test cases [15–17]. The intuition that underlies the choice and the definition of the ESE evolutionary operators is that the path conditions sample the possible program paths more effectively than concrete inputs, since there is a one-to-one mapping between distinct path conditions and distinct program paths, while the possible input values may unevenly distribute across the classes of inputs that trigger distinct program paths, up to some paths being executable only with singular and hard to be sampled inputs. To the best of our knowledge, the ESE algorithm is unique in exploiting the synergy between symbolic execution, which contributes the ability to identify the execution conditions that characterize the program paths, and a memetic path selection heuristic, which contributes to the ability of heuristically (i.e. not exhaustively) exploring huge path spaces. In the experiments reported in this paper, we investigated our hypotheses about the ESE evolutionary operators by comparing ESE with the search-based test generator EvoSuite that uses evolutionary operators based on manipulating concrete inputs and that we purposely adapted to use the same fitness function as ESE.

We presented ESE for the first time at the International Symposium on Software Reliability Engineering (ISSRE), in 2018 [18]. This paper properly extends our previous work in several significant ways. First, since the previous paper, we re-engineered the ESE prototype that we use for the experimental evaluation of the approach. The new prototype solves many performance issues that, at the time of the previous paper, were hampering us to report conclusive results for some experiments. Thus, this paper reports refined empirical data (4) that provide a more comprehensive picture of the potential of the ESE approach than in the previous paper. Second, this paper includes a new set of replications of the experiments that investigate the sensitivity of the ESE approach with respect to the main parameters of the memetic algorithm, namely, the number of individuals (path conditions) in each generation of the genetic algorithm, the probability of applying mutations to the individuals, the amount of good individuals that survive across subsequent generations (elitism) and the frequency of calls to the local search step of the algorithm. Finally, this paper originally discusses and experiments the merit of the evolutionary operators of ESE, based on symbolic path conditions, with respect to the alternative choice of using the same fitness function as ESE, but using traditional dynamic operators that manipulate concrete inputs.

The paper is organized as follows. Section 2 motivates our research by exemplifying the existing techniques, WISE and SPF-WCA, and their limitations on a set of sample programs. Section 3 presents our novel technique for WCET testing in detail. Section 4 discusses a set of experiments that evaluate our technique both in absolute terms and in comparison with WISE and SPF-WCA. Section 5 acknowledges the related work in the field. Finally, Section 6 summarizes our conclusions and outlines directions of further research.

2. MOTIVATION

This section discusses a set of examples that highlight the limitations of the state-of-the-art WCET testing techniques, thus motivating the new technique proposed in this paper.

DOI: 10.1002/stvr

Listing 1: `is_palindrome`

```
"""
Decides if the given list is a palindrome.

Worst case: a palindrome list.
"""
def is_palindrome(l):
    for i in range(len(l)):
        if l[i] != l[len(l) - 1 - i)]:
            return False
    return True
```

The two phases symbolic execution approach that we surveyed in the introduction was first proposed in the technique WISE [8]. We illustrate WISE with reference to the program `is_palindrome` in Listing 1 that inspects an input list to answer whether it is palindromic. For the input lists of a fixed size, a WCET test case amounts to executing the program with a list that is indeed palindromic, thus causing the maximum number of iterations of the loop in the program. To find a WCET test case for this program, WISE starts with the exhaustive symbolic execution of the program in a restricted input scope that suffices to limit the feasible program paths to a manageable amount. The way to express the restriction tightly depends on the characteristics of the inputs of the program under analysis and must be thus indicated by a test analyst in the general case. If the input is a list of primitive values, as in the case of the program in Listing 1, WISE can work by bounding the size of the input list to few items.

For the program in Listing 1, considering only lists with five items limits the number of possible execution paths to three feasible (and three infeasible) paths. The feasible paths correspond to the executions in which the program exits the loop during the first or the second iteration, or completes all the five iterations, respectively. Symbolic execution captures these paths with the path conditions (i) $\alpha_1 \neq \alpha_5$, (ii) $\alpha_1 = \alpha_5 \wedge \alpha_2 \neq \alpha_4$ and (iii) $\alpha_1 = \alpha_5 \wedge \alpha_2 = \alpha_4$, respectively, being $\alpha_{1..5}$ symbolic values that represent the five items in the input list. The last of these path conditions characterizes the palindromic lists, for example, $[1, 2, 0, 2, 1]$. The infeasible paths correspond to the paths in which the program would exit the loop during the third, the fourth or the fifth iteration, which is impossible in all three cases because the third list item cannot be different from itself, and the conditions to exit at the fourth, or the fifth iteration contrast with the assumptions made at earlier iterations. Symbolic execution identifies that these three paths are infeasible because they map to unsatisfiable path conditions, for instance, the path condition of the path that exits the loop at the fourth iteration is $\alpha_1 = \alpha_5 \wedge \alpha_2 = \alpha_4 \wedge \alpha_3 = \alpha_3 \wedge \alpha_4 \neq \alpha_2$, in which the second and the fourth conditions are mutually contradictory.

For the program of Listing 1, WISE would essentially work as follows. It symbolically executes the program with respect to a small input list, for example, with the input list bounded to five items as above mentioned. Then, for each feasible path yielded by symbolic execution, it measures the number of instructions traversed in the path and selects the path that executes the highest amount of instructions. For the lists with five items, it obtains that the worst path is the one that completes five iterations of the loop in the program. Next, it observes that symbolic execution consistently selected the *false* branch of the if-statement inside the loop, and thus it infers that forcing symbolic execution to select this branch consistently guides the analysis throughout the worst path of the program. Indeed, using this *guidance policy*, WISE can efficiently analyze the program of Listing 1 for arbitrarily large input lists, because the guidance policy steers symbolic execution to explore only one path, and exactly the worst path of the program. For instance, considering input lists with 100 items, the above guidance policy leads WISE to explore the single path whose path condition is $\alpha_1 = \alpha_{100} \wedge \alpha_2 = \alpha_{99} \wedge \alpha_3 = \alpha_{98} \wedge \cdots \wedge \alpha_{50} = \alpha_{51}$ and then generates the WCET test case by solving this condition to actual values.

DOI: 10.1002/stvr

Listing 2: `alternate_0`

```
"""
Iterates an expensive task based on the values of the list.

Worst case: a list that alternates zeros and non-zero values.
"""
def alternate_0(l):
    for i in range(len(l)):
        if l[i] == 0:
            check = 1 - (i % 2)
        else:
            check = i % 2

        if check != 0:
            execute_expensive_task()
```

The SPF-WCA approach [9] extends WISE, synthesizing sophisticated guidance policies in which the decisions to make at the relevant decision points must not be consistently the same, as in the above example, but may vary depending on the history of decisions made during symbolic execution. Listing 2 exemplifies the program `alternate_0` for which SPF-WCA improves on WISE. For this program, the WCET test cases amount to executing the program with lists that contain zero and non-zero numbers at even and odd locations, respectively, thus making function `execute_expensive_task()` execute at all iterations of the loop in the program. To execute the worst case path of the program in Listing 2, symbolic execution must proceed in strict alternation through the *true* and *false* branches of the first if-statement inside the loop. In this case, WISE would synthesize an ineffective guidance policy, since it just observes that both decisions shall be allowed at that if-statement, and this results in providing no actual guidance. Instead, SPF-WCA is able to compute an effective policy that guides symbolic execution to select the *true* branch if it selected the *false* branch at the previous traversal of the if-statement and the vice versa. SPF-WCA can be instantiated to use decision histories of any size, but for this example, it suffices to inspect only one previous decision.

In this paper, we question the general validity of the fundamental assumption that underlies both WISE and SPF-WCA, that is, we question that the analysis of the program with restricted inputs can always unveil a regular worst case behaviour. We argue that in the general case, the worst program path can either be irregular across increasingly large input bounds or even correspond to different program behaviours for different boundings. Below, we provide examples of these observations.

Listing 3: `depth_first_search`

```
"""
Determines if the vertex "finish" is reachable from the vertex "start"
in the given graph.

Worst case: a fully connected directed graph, except for the vertex
"finish" that is reachable only from the "start" vertex of which it is
the last neighbor.
"""
def dfs(graph, start, finish):
   return do_search(graph, start, finish, []);

def do_search(graph, current, finish, visited):
   if current == finish:
       return true;

   visited.append(current)

   for neighbor in graph.outgoing(current):
       if not neighbor in visited:
           if do_search(graph, neighbor, finish, visited)
               return true;
   return false;
```


DOI: 10.1002/stvr

Listing 3 shows the `dfs` program for which the branch sequences of the worst path cannot be captured with WISE and SPF-WCA. The program implements a depth-first visit to find if there is a path that connects two vertices 'start' and 'finish' in a graph. The worst case happens with a directed graph in which all vertices but *finish* are fully interconnected, and the vertex *finish* is connected only to *start* and listed as its last neighbour. A graph with this structure requires the program to visit all the vertices in the graph, and the maximum amount of edges for each vertex, while requiring to backtrack from the full visit to find the wanted path. For this program, in any input restriction in which the input graph has a fixed amount of vertices, the worst case graph leads symbolic execution through a sequence of decisions such that (i) the worst path includes both at least a *true* and at least a *false* outcome for all decision points in the program, thus making WISE be unable to identify an effective guidance policy, and (ii) the decision history needed to identify the right decision at each point of the sequence is a decision history of different size for each considered input restriction, which makes SPF-WCA be unable to identify a guidance policy. For instance, with reference to graphs of four vertices, the worst case path traverses the first if-statement in the `do_search` function with the sequence of decisions $\langle false, false, false, true \rangle$, where the first true decision happens after three *false* decisions, while with reference to graphs of five vertices the sequence should be $\langle false, false, false, false, true \rangle$, where the first *true* decision happens after four *false* decisions.

Listing 4: `memory_fill`

```
"""
Copy non-zero values in a buffer of 16 cells.

Worst case: a list of non-zero values.
"""
def memory_fill(l):
    memory = [0] * 16
    free = 16

    if len(l) <= free:
        for i in range(len(l)):
            memory[i] = l[i]
        return

    for i in range(len(l)):
        if l[i] != 0:
            free -= 1
            if free >= 0:
                memory[free] = l[i]
```

Listing 4 shows the `memory_fill` program for which the worst path cannot be observed in a suitable input restriction. The program copies a list of inputs into a fixed size buffer with 16 cells, aiming at copying the largest set of non-zero values that fit in the buffer. The program handles two cases: (i) if the amount of inputs is less than the size of the buffer, the program optimizes its performance by making a straight copy of all inputs; (ii) otherwise, it inspects the value of each input and copies only the largest fitting set of non-zero values. The WCET test case of this program is an input larger than the buffer and comprised non-zero values only since each non-zero value makes the program execute the longest block of instructions. Interestingly, this WCET test case reveals a performance bug, since the program wastes time to scan inputs beyond the last one that fits in the buffer. This turns out being also a security vulnerability since an attacker might feed the program with enormous inputs to cause a denial of service.

The worst path of the program showed in Listing 4 cannot be observed when bounding the input list to less than 17 items, since with these inputs the program executes only the part of the algorithm that makes a straight copy of all inputs. Similarly, a scope with 17 input items leads to 217 possible execution paths, which cannot be exhaustively analyzed with symbolic execution in reasonable time. Indeed, for this program, both WISE and SPF-WCA would limit the restricted analysis to at most 16 input items, thus failing to analyze the part of the code that corresponds to the worst case path.

DOI: 10.1002/stvr

Another similar example is reported in the seminal paper of WISE [8], in which the authors discuss a third-party implementation of `quicksort` (taken from the Java Development Kit (JDK) 1.5) that they considered as an experimental subject. They notice that the JDK-1.5 quicksort included the optimization of using a median-of-9 pivot when sorting arrays with more than 40 items and a median-of-3 pivot when sorting smaller arrays. Thus, they downgraded the implementation to always use a median-of-3 pivot, acknowledging that otherwise, the guidance policy computed in the small scope would be inconsistent with the behaviour of the program with more than 40 inputs.

The next section presents our technique for WCET testing whose core novelty is to search for the worst case behaviour of a program by heuristically analyzing it directly in the target scope, thus not suffering the above issues.

3. EVOLUTIONARY SYMBOLIC EXECUTION

This section presents our novel technique for WCET testing that works by combining symbolic execution with an evolutionary path selection strategy based on a memetic search algorithm. We refer to this technique as Evolutionary Symbolic Execution (ESE). ESE considers the set of the feasible program paths as the search space, through which it steers symbolic execution to explore a sample of incrementally selected program paths, up to ultimately identifying a program path that reveals the worst case execution time of the program. Then, it generates the WCET test case by exploiting the symbolic representation of this path. The core of ESE is a memetic algorithm that incrementally selects the program paths to explore during the search, with the aim of identifying the WCET behaviour as quickly as possible.

3.1. Overview of the ESE algorithm

Algorithm 1 outlines the workflow of our ESE technique in pseudocode, which is also graphically illustrated in Figure 1. The input comprised the program under test and a set parameters to configure the algorithm as we describe below. The algorithm starts with generating an initial set (referred to as *population*) of randomly selected feasible paths (Line 1). The details of the algorithm RANDOMPATHSAMPLING(), which underlies this step, are presented in Section 3.2. In a nutshell, this step steers symbolic execution with a random path selection strategy, stopping after visiting a predefined amount of *popsize* program paths. We refer to each symbolically analyzed path as an 'individual' of the current population. An individual stores the path condition computed for the corresponding path, and the counting of the instructions traversed while symbolically executing that path. As a large majority of techniques for WCET analysis [8, 9], we assume that the number of instructions that are executed along a program path is a good proxy of the execution time of the program along that path. In our search-based algorithm, the function that associates the explored program paths, that is, the individuals, with the corresponding amount of executed instructions plays the role of the 'fitness function' that the algorithm aims to maximize.

Next, the algorithm continues with symbolically executing additional program paths, building on the knowledge of the path costs observed in the current population, to select program paths that might improve on the fitness of the current ones. Working in the style of genetic algorithms [13, 14], we specify the path selection strategy that we use in this step in the form of a crossover mechanism (Lines 4–7) that exploits the path conditions of (pairs of) already explored paths (selected as the *parent pairs* at Line 4) to steer symbolic execution to traverse new feasible program paths (the *children* at Line 6) that might improve on the fitness of the parents. The SELECTION() called at Line 4 corresponds to the classical selection operator of genetic algorithms that picks random individuals from the current population, with the probability of picking each individual being proportional to its current fitness. Section 3.3 presents the details of the algorithm CROSSOVER() (Line 6) that is the core of the crossover mechanism. It consists in enforcing symbolic execution to comply with subsets of execution conditions selected and combined from the parents' path conditions. We build on the intuition that the execution conditions of the parents may convey costly subpaths to propagate in the children, while the children may still include other (possibly newly explored) subpaths that are not constrained by those execution conditions.

DOI: 10.1002/stvr

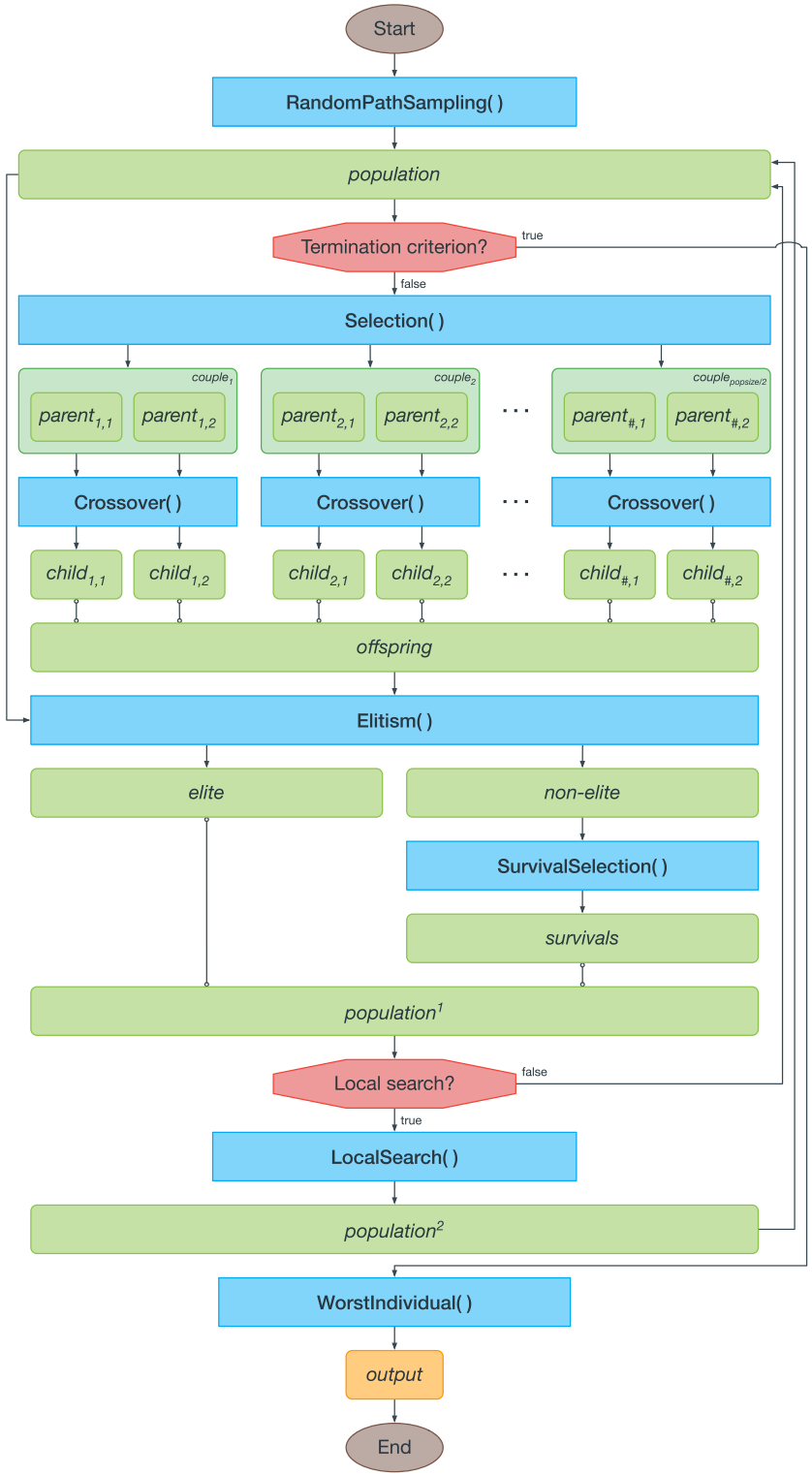

Figure 1. Workflow of the ESE algorithm 1

DOI: 10.1002/stvr

Then, our algorithm consolidates (Line 11) the results of the crossover by shaping a new population that includes a small set of the currently fittest (the worst program paths) individuals (ELITISM(), Line 9) and a fitness-biased random selection of the others (SURVIVALSELECTION(), Line 10) in the relative amounts specified by parameter *elitesize* and its complement, *popsize–elitesize*, respectively. The algorithm iterates through this process within a predefined testing budget (TESTINGBUDGETEXHAUSTED(), Line 3) that can be specified either as a timeout or a maximum amount of iterations.

Algorithm 1 The evolutionary symbolic execution algorithm

Input *:program*, the program under test
popsize, the number of program paths that shall be explored at each iteration
elitesize, the number of best-fit program paths that shall be retained at the next iteration
mprob, the probability of applying the mutation operator during crossover
lsperiod the number of iterations before calling the local search algorithm
lsattempts the number of optimization attempts in the local search algorithm
Output :a test case that executes the worst program path found in program

1 *population* ← RandomPathSampling*(program, popsize)*
2 *generations* ← 0
3 **While** TESTINGBUDGETEXHAUSTED() **do**
4 *parentpairs* ← Selection *population*
5 **for** $parent_1$, $parent_2$ *in parentpairs* **do**
6 $child_1$, $child_2$ ← CROSSOVER *(program,* $parent_1$*,,* $parent_2$*,, mprob)*
7 *offspring* ← *offspring* ∪ {$child_1$, } ∪ {$child_2$, }
8 *population* ← *population* ∪ *offspring*
9 *elite, non-elite* ← ELITISM *(population, elitesize)*
10 *survivals* ← SURVIVALSELECTION *(non-elite, popsize – elitesize)*
11 *population* ← *elite* ∪ *survivals*
12 *generations* ← *generations* + 1
13 **if** generations % lsperiod = 0 **then**
14 *worst* ← WORSTINDIVIDUAL *(population)*
15 *worst'* ← LOCALSEARCH *(program, worst, lsattempts)*
16 *population* ← (*population* –{*worst*}) ∪ {*worst'*}
17 **return** SOLVEPATHCONDITIONTOTESTINPUTS(WORSTINDIVIDUAL*(population)*)
18 **Function** TESTINGBUDGETEXHAUSTED()
19 **return***whether we exhausted the testing budget, because of having either executed the maximum amount of iterations or expired the timeout*
20 **Function** SELECTION *(population)*
21 **return** *a set of |population| /2 randomly selected pairs* $(i_1; i_2)$*, with* $i_1, i_2 \in$ *population. At each random pick, the probability of selecting a given* $i \in$ *population is proportional to its current fitness*
22 **Function** ELITISM *(population, elitesize)*
23 **return** *the elitesize fittest individuals (the worst program paths) out of population*
24 **Function** SURVIVALSELECTION *(population, amount)*
25 **return** *a set of amount individuals picked at random out of population. At each random pick, the probability of selecting a given* $i \in$ *population is proportional to its current fitness*
26 **Function** WORSTINDIVIDUAL *(population)*
27 **return** *the fittest individual (worst program path) out of population*
28 **Function** SOLVEPATHCONDITIONTOTESTINPUTS*(individual)*
29 **return** *test inputs generated by solving the path condition of individual with a constraint solver*

DOI: 10.1002/stvr

We observe that the SURVIVALSELECTION() step at 10 of Algorithm 1 consists of the same type of computation as the SELECTION() step at 4. We keep them with separate names to emphasize the different purposes of the two steps in the context of Algorithm 1: SELECTION() aims to select pairs of individuals to feed the crossover, while SURVIVALSELECTION() propagates individuals across subsequent generations.

At regular intervals, that is, when the number of iterations is multiple of a predefined *lsperiod* period value (Line 13), the algorithm accomplishes a local search, trying to further optimize the worst program path computed so far (Lines 14–16). This step indeed classifies the algorithm as 'memetic', and not simply as 'genetic'. Section 3.4 presents the algorithm LOCALSEARCH() (Line 5) in detail. It consists in a hill-climbing strategy that incrementally negates a single random condition out of the ones in the path condition of the current worst individual, in the attempt to reveal suboptimal decisions that may further worsen the execution cost when inverted. After a fixed amount of attempts, specified by the parameter *lsattempts*, it returns the worst individual identified along the process, or the initial individual unchanged if all attempts failed, which replaces the previous worst individual thereon (Line 16).

Upon exhausting the testing budget, the algorithm returns a test case that it obtains by solving (with a constraint solver, e.g., †Z3[‡][19]) the path condition of the worst individual to concrete inputs that make the worst program path execute (Line 16).

Section 4 summarizes all the configuration parameters of the algorithm and their concrete values in the context of our experiments.

3.2. Exploring and representing program paths

Our search-based algorithm computes the initial population of candidate solutions by exploring a random sampling of feasible program paths with symbolic execution. Algorithm 2 and Algorithm 3 specify this computation in pseudocode.

Algorithm 2, that is, RANDOMPATHSAMPLING(), initializes a population with *npaths* individuals by iterating (Algorithm 2, Line 2) as follows. It analyzes the program under test with symbolic execution, to compute the path condition and the number of instructions of a randomly chosen path (Algorithm 2, Line 3), and then encodes these results as an individual object, that is, a candidate solution, of the search algorithm (Algorithm 2, Line 4). The conjunctive formula that comprises the path condition is the 'chromosome' representation of the individual, whereas the atomic constraints represent the 'genes'. The number of instructions in the program path is the measurement of the fitness of the individual.

Algorithm 2 RANDOMPATHSAMPLING *(program, npaths)*

input :program, the program under test npaths, the number of paths to be randomly sampled

output :a population of randomly sampled program paths (individuals)

```
1  population ← Ø
2  for i ← 1 to npaths do
3     pc, instrs ← SYMBOLICEXECUTION(program)
4     individual ← INDIVIDUAL(pc, instrs)
5     population ← population ∪ {individual}
6  return population
7  Function INDIVIDUAL (pc, instrs)
8     return an individual that represents a program path as a structure that contains the path
      condition and the amount of instructions of the program path
```

[‡]†Z3 is a SMT constraint solver. Given a formula in a supported logic theory, e.g., linear formulas over integers, non linear formulas over reals, and so forth, it can either prove that the formula is satisfiable by computing a satisfying assignment of the variables in the formula, or yield a proof that the formula is unsatisfiable.

DOI: 10.1002/stvr

Algorithm 3, SYMBOLICEXECUTION(), specifies the exploration of a randomly chosen feasible program path with symbolic execution. It starts with building an initial symbolic state in which the program inputs are assigned to symbolic values (Algorithm 3, Line 2), and then iteratively computes the possible successor states as in classic symbolic execution, that is, by manipulating the symbolic representation of the current state according to the semantics of the current program statement and updating the program counter to point to the next statement to be executed (Algorithm 3, Line 4).

The function SYMBOLICEXECUTIONSTEP() at Line 13 formalizes the semantics of executing a symbolic execution step with reference to simplified programs that include only assignment statements of the form `x:=e;`ℓ' and conditional jump statements of the form `if(c)` ℓ' `else` ℓ'', where `x` generically denotes a program variable, `e` and `c` denote code expressions over program variables and ℓ' and ℓ'' denote the program locations reached as result of the execution of the statements. A symbolic state is a triple $\langle \ell, vv, pc \rangle$, where ℓ is the current program location during symbolic execution, vv are the current, possibly symbolic, values of the program variables, and pc (the path condition) is the set of executability conditions assumed while symbolically executing the program up to the current location. For instance, in the initial state $\langle \ell_0, vv_0, pc_0 \rangle$, ℓ_0 is the entry point of the program, vv_0 associates the program inputs to unconstrained, distinct symbolic values and pc_0 is simply $true$ because reaching the entry point is always reachable. From a generic state $\langle \ell, vv, pc \rangle$, symbolic execution progresses by executing the program statement specified at the location ℓ: If the statement at ℓ is an assignment `x:=e;` ℓ', then we reach a new state $\langle \ell', vv', pc \rangle$, where ℓ' is the next program location, vv' is the same as vv for all variables but `x`, which is now assigned according to the result of the executed assignment, and the path condition is unchanged. If the statement at ℓ is a

Algorithm 3 SYMBOLICEXECUTION(*program*)

input :*program*, the program under test
output :the path condition and the number of instructions of a randomly selected program path

1 *instrs* ← 0
2 *state* ← INITIALSYMBOLICSTATE*(program)*
3 **while** ¬ ISENDSTATE*(state)* **do**
4 *successors* ← SYMBOLICEXECSTEP*(state)*
5 *state* RANDOMSELECTION*(successors)*
6 *instrs* ← *instrs* + 1
7 *pc* ← PATHCONDITION*(state)*
8 **return** *pc, instrs*
9 **Function** INITIALSYMBOLICSTATE*(program)*
10 **return** *the initial symbolic state where all inputs are assigned as unconstrained symbolic values*
11 **Function** ISENDSTATE*(state)*
12 **return** *whether this state corresponds to having reached the end of a program path*
13 **Function** SYMBOLICEXECSTEP*(state)*
14 **Let** *state* ≡ *(ℓ vv, pc)* ▷ where ℓ is the program location reached at state, *vv* are the current symbolic values of the program variables at state, and pc is the current path condition at state
15 $succ \leftarrow \begin{cases} \{(\ell', vv\,[\mathtt{x} \leftarrow [\![\mathtt{e}]\!]_{vv}], pc)\} & \text{if } \mathrm{st}(\ell) \equiv \mathtt{x} := \mathtt{e};\ell' \\ \{(\ell', vv, pc \wedge [\![\mathtt{c}]\!]_{vv}), (\ell'', vv, pc \wedge \neg[\![\mathtt{c}]\!]_{vv})\} & \text{if } \mathrm{st}(\ell) \equiv \mathtt{if(c)}\ell'\mathtt{else}\,\ell'' \end{cases}$
▷ where: st(ℓ) is the statement that the program executes at location ℓ, $[\![.]\!]$ *vv* denotes the evaluation of a program expression with respect to the values *vv* of the program variables, and ℓ', ℓ'' are other program locations that can be reached as result of the execution of the current statement
16 **return** *succ.* ▷ the successors of state by computing a single symbolic execution step
17 **Function** RANDOMSELECTION*(states)*
18 **return** *a random state out of states*
19 **Function** PATHCONDITION*(state)*
20 **return** *the path condition of this state*

DOI: 10.1002/stvr

conditional jump `if(c)` ℓ' `else` ℓ'', then we progress to two possible states, either $\langle \ell', vv, pc' \rangle$ or $\langle \ell'', vv, pc'' \rangle$, where ℓ' and ℓ'' are the program locations of the then-branch and else-branch of the statement, respectively, the program variables are unchanged, and the path conditions pc' and pc'' are obtained by conjoining the path condition pc to the positive and negative evaluation of the conditional of the executed statement, respectively.

To explain the use of symbolic execution in our algorithm, the key observation is that symbolic execution step may yield either a single successor state, when executing non-branching statements in the program, like the statements that correspond to assignments of variables, or more than one successor states, when executing branching statements, like the statements that evaluate conditions at the decision points in the program. We refer to the standard embodiment of symbolic execution that relies on a constraint solver to incrementally check the reachability of the successor states, by testing the satisfiability of the corresponding path condition formulas, and discards any unreachable successor state. When the solver confirms more than a successor state, our algorithm chooses a random state out of those and continues the analysis on that state only (Algorithm 3, Line 5), until meeting the end of a program path (Algorithm 3, Line 3). Our algorithm finally returns (Algorithm 3, Line 8) the path condition of the analyzed program path. We remark that, the returned path condition can be solved to a satisfying assignment with a constraint solver, to obtain concrete input values for the corresponding program path in a test case.

This procedure guarantees that each run of symbolic execution according to Algorithm 3 explores a single program path, which is feasible based on the outcomes of the constraint solver, and which is randomly selected at each decision point where the execution might proceed through multiple distinct feasible paths. As byproduct, the procedure measures the execution cost of the analyzed path as the number of steps executed while symbolically analyzing the path (Algorithm 3, Line 6).

3.3. Genetic operators

We bootstrap our search-based algorithm with the random sample of program paths collected as discussed in the previous section, and then proceed with steering symbolic execution to explore additional program paths, by alternating between global and local search phases. The global search

Algorithm 4 CROSSOVER(*program*, $parent_1$, $parent_2$, *mprob*)

input :*program*, the program under test
$parent_1$ and $parent_2$, two individuals of the current population
mprob, the probability of applying the mutation operator during crossover
output :two new individuals generated by crossing the path conditions of $parent_1$ and $parent_2$

1 $pc_i \leftarrow$ PATHCONDITION*(parenti$_i$)*, $\forall\, i \in \{1,2\}$
2 $len_i \leftarrow$ the amount of conditionals in pc_i, $\forall\, i \in \{1,2\}$
3 $cut_i \leftarrow$ a random integer between 1 and len_i, $\forall\, i \in \{1,2\}$
4 $\text{pre}_1 \leftarrow pc_1\ [0{:}\ cut_1] \wedge pc_2\ [cut_2 : len_2]$
5 $\text{pre}_2 \leftarrow pc_2\ [0{:}\ cut_2] \wedge pc_1\ [cut_1 : len_1]$
6 *children* $\leftarrow \emptyset$
7 **for** *pre* $\in \{\text{pre}_1, \text{pre}_2\}$ **do**
8 **if** *random number in* [0, 1] < *mprob* **then**
9 *pre* $\leftarrow$ MUTATERANDOM*(pre)*
10 *pc, instrs* $\leftarrow$ SYMBOLICEXECUTIONPRE*(program, pre)*
11 *children* $\leftarrow$ *children* $\cup$ INDIVIDUAL*(pc, instrs)*
12 **return** *children*
13 **Function** MUTATERANDOM*(pre)*
14 **return** *pre' ← pre with up to 10% of the conditionals removed*
15 **Function** PATHCONDITION*(state)*
16 ▷ ... `as in Algorithm 3`
17 **Function** INDIVIDUAL*(pc, instrs)*
18 ▷ ... `as in Algorithm 2`

DOI: 10.1002/stvr

phase is a genetic algorithm that exploits the information in the current population of candidate solutions. The local search phase focuses on the best currently identified solution only. This subsection describes our genetic algorithm, while we present the algorithm of the local search phase in the next subsection.

Our genetic algorithm fosters the exploration of program paths that may probabilistically include and combine both high-cost subpaths that were already observed in some current individuals, and other, possibly new randomly explored subpaths of the program. The core of this computation is done by the crossover operator of the genetic algorithm.

Algorithm 4, that is, CROSSOVER(), specifies the crossover operator in pseudocode. The crossover works on pairs of individuals selected from the current population, here denoted as the inputs $parent_1$ and $parent_2$. As we already commented earlier in this section (Algorithm 1, Line 4) the selection of $parent_1$ and $parent_2$ is accomplished as a random pick according to a non-uniform distribution, such that the individuals with higher fitness have higher probability of being selected than the ones with lower fitness. This type of selection mechanism is standard in genetic algorithms, and suits our algorithm with no particular adaptation, thus we do not discuss it further. We remark only that our selection operator enforces $parent_1$ and $parent_2$ to be different individuals of the current population.

Our crossover fosters symbolic execution to explore at most two additional program paths, and enforces these paths to comply with partial sets of the execution conditions excerpted from the path conditions of the two parents, $parent_1$ and $parent_2$, thus possibly replicating subpaths of these individuals. The algorithm (i) synthesizes two new conditions pre_1 and pre_2 by combining the path conditions of the two parents (Algorithm 4, Lines 1–5), (ii) mutates each condition $pre \in \{pre_1, pre_2\}$ with some probability (Lines 8–9) and (iii) exploits each condition $pre \in \{pre_1, pre_2\}$ with symbolic execution to collect the offspring individuals that the crossover returns as result (Lines 10–11).

To synthesize the new conditions pre_1 and pre_2, we cut the path conditions of the two parents at random cutpoints and join the first part of the path condition of $parent_1$ with the second part of the path condition of $parent_2$ (Algorithm 4, Lines 1–4) and the vice-versa (Algorithm 4, Line 5). In this phase, the algorithm relies on the knowledge that symbolic execution yields conjunctive path conditions and thus regards the path conditions simply as lists of clauses. Then, with predefined probability *mprob*, we may mutate the new conditions (both pre_1 and pre_2, either one, or none of

Algorithm 5 SYMBOLICEXECUTIONPRE(*program*, *pre*)

input :*program*, the program under test
pre, a set of preconditions
output :the path condition and the number of instructions of a randomly selected program path whose path condition does not contradict *pre*

```
1  instrs ← 0
2  state ← INITIALSYMBOLICSTATE(program)
3  while ¬ ISENDSTATE(state) do
4     successors ← SYMBOLICEXECSTEP(state)
5     successors ← PRUNEUNSATSTATES(successors, pre)
6     if successors = Ø then
7        throw SymbolicExecutionException
8     state ← RANDOMSELECTION(successors)
9     instrs ← instrs + 1
10 pc ← PATHCONDITION(state)
11 return pc, instrs
12 Function PRUNEUNSATSTATES(states, pre)
13    return states – {s ∈ states, s.t. pre ^ PATHCONDITION(s) is unsatisfiable}
14 Function INITIALSYMBOLICSTATE, ISENDSTATE, SYMBOLICEXECSTEP,
      RANDOMSELECTION,PATHCONDITION
15       ▷ ... as in Algorithm 3
```


DOI: 10.1002/stvr

them) by removing some inner conditions chosen at random (Algorithm 4, Lines 8–9). The actual mutation algorithm removes from a minimum of a single condition up to a maximum of 10% of the inner conditions. After each removal, it decides with even probability either to stop or continue with removing a further condition; it stops necessarily after removing the maximum number of conditions.

Algorithm 5, that is, SYMBOLICEXECUTIONPRE(), specifies the symbolic exploration of a feasible program path that complies with a set of conditions *pre* synthesized in the crossover algorithm. The symbolic execution algorithm mimics all steps of Algorithm 3 with the only additional behaviour of pruning the symbolic states that are incompatible with the precondition *pre* (Algorithm 5, Lines 5–7). We rely on the constraint solver to decide whether the path condition of a current symbolic state is satisfiable in conjunction with *pre* and prune the symbolic states for which the solver returns an unsatisfiability verdict (Algorithm 5, Line 5). If all successor states happen to be incompatible with *pre* (Algorithm 5, Line 6), the symbolic execution algorithm terminates with an exception (Algorithm 5, Line 7) indicating that the precondition prevents the execution of any feasible path of the program. The crossover does not generate the individual in this latter case—in Algorithm 4, for simplicity, we do not show this exceptional behaviour explicitly.

This procedure guarantees that each (unexceptional) run of symbolic execution according to Algorithm 5 explores a single feasible program path, which complies with subsets of the execution conditions of the parent individuals, thus likely contains subpaths that belong also to those individuals, and which is randomly selected at any decision points with multiple paths that are not constrained by the precondition *pre*.

3.4. Local search

At regular intervals, our algorithm accomplishes a local search phase in which it tries to make small focused changes to the individual that has the maximum cost in the current population, trying to further optimize that candidate solution. In this phase, the algorithm proceeds in the style of the 'hill climbing' algorithm, that is, incrementally changing single elements of the solution and accepting the changes that result in better solutions. The changes consist in exploring program paths that differ from the current one for the outcome at a single decision point.

Algorithm 6, that is, LOCALSEARCH(), specifies the local search algorithm in pseudocode. The input *individual* denotes the individual with maximum cost in the current population at the beginning of the local search. The algorithm performs a fixed amount of iterations (Line 1). At each iteration,

Algorithm 6 LOCALSEARCH(*program*, *individual, lsattempts*)

```
input  :program, the program under test
        individual, an individual of the current population
        lsattempts the number of optimization attempts in the local search algorithm
output  :the worst individual identified by applying incremental local mutations
        to the one in input
1   for attempt ← 1 to lsattempts do
2         pre ← PATHCONDITION(individual)
3         pre ← INVERTRANDOMCONDITION(pre)
4         pc, instrs ← SYMBOLICEXECUTIONPRE(program, pre)
5         if instrs > individual.instrs then
6          individual ← INDIVIDUAL(pc, instrs)
7   return individual
8   Function INVERTRANDOMCONDITION(pre)
9       return pre' ← pre after negating a randomly chosen conditional
10  Function PATHCONDITION
11      ▷ ... as in Algorithm 3
12  Function INDIVIDUAL
13      ▷ ... as in Algorithm 2
```


DOI: 10.1002/stvr

it inverts a condition chosen at random out of the path condition of the current individual, that is, it replaces that condition with its logical negation (Lines 2–3). Then, it computes a new individual in a similar fashion as we explained for the crossover operator (Line 4). If the new individual has higher execution cost than the current one, the local search replaces the current with the new individual (Lines 6–6), since the latter is closer to the optimal solution than the former, and iterates to further optimize the new individual. Otherwise, the algorithm continues without changing the current individual. The algorithm returns the individual with the highest cost identified throughout this process.

The intuition that underlies the local search phase is that, when the global search computes some solution that is close to the optimum, likely, it is a program path that mimics the worse program path except for a small set of branch decisions. In this situation, the crossover is generally ineffective to find the missing optimizations without altering other parts of the path. Conversely, trying a punctual exploration of the possible changes is more likely to succeed. The local search is also effective to identify subpaths with regular behaviour, for example, a subpath in which all decisions at an *if* statement must regularly take the same branch (or alternated branches) for a given amount of subsequent evaluations of that decision point. Although the global search may succeed to identify a majority of the needed decisions, the fully regular sequence may appear like a singularity in the search space. The local search may incrementally spot the suboptimal decisions and fix them.

4. EXPERIMENTS

We implemented a prototype of ESE for Python programs (ESE_{py}) and used it to experimentally investigate the effectiveness of ESE with respect to a set of sample programs for inputs of increasing size. In particular, our experiments address the following research questions:

1. What is the sensitivity of ESE with respect to the main parameters of the ESE algorithm?
2. Does ESE steer symbolic execution towards generating WCET test cases?
3. How does ESE compare with the state-of-the-art approaches for WCET testing, that is, WISE and SPF-WCA?
4. What is the contribution of the evolutionary operators of ESE, which require computation and manipulation of symbolic path conditions, over just using the WCET fitness function of ESE with a purely dynamic search-based testing algorithm, for example, EvoSuite?

In RQ1, we look at the sensitivity of ESE with respect to the main parameters of the evolutionary algorithm, that is, the size of the population, the amount of the best individuals (elite) that survive across subsequent generations, the frequency of mutations, and the frequency of the local search steps.

In RQ2, we are interested in confirming whether ESE significantly outperforms the baseline strategies of steering symbolic execution of analyzing the feasible program paths in either depth-first or random order, respectively. We study RQ2 by using a proper configuration for ESE, determined based on the results that we observed while studying RQ1.

In RQ3, we investigate the validity of the research hypothesis that we illustrated with the examples in Section 2, where we discussed that the approaches WISE and SPF-WCA work well for programs where the worst-case behaviour generalizes with regularity for inputs of increasing size, but perform poorly in the cases in which there is no such regularity, while ESE can significantly outperform WISE and SPF-WCA in these latter cases.

Finally, in RQ4, we compare ESE with a purely dynamic search-based algorithm, obtained by adapting EvoSuite to use the same fitness function, that is, the count of executed instructions, as ESE. We aim at investigating the specific merit of the novel evolutionary operators of ESE, defined with respect to symbolic path conditions, in comparison to using evolutionary operators that directly manipulate the program inputs.

Below, we introduce our prototype of ESE and discuss the setting and the results of our experiments.

DOI: 10.1002/stvr

4.1. Evolutionary symbolic execution prototype

Our prototype ESE$_p y$ is implemented in Python, based on a purposely designed symbolic executor for Python programs. The symbolic executor relies on the Z3 constraint solver [19], and works by instrumenting that inputs data of the program under test, to make the Python interpreter handle the inputs as symbolic values (similarly to the work of Saxena et al. on symbolic execution of JavaScript programs [20]). It can currently handle input data that consist of integers and bounded collections of integers.

4.2. Experiments setting

Our experiments challenge ESE$_p y$ to generate WCET test cases for the sample programs in Table I: `alternate_0`, `is_palindrome`, `dfs`, and `memory_fill` are the programs that we already discussed in Section 2, `merge_sort` and `quicksort_jdk` are the popular sorting algorithms (in particular `quicksort_jdk` is the JDK-1.5 quicksort algorithm that switches from a mid-array pivot for arrays with less than seven items, to a median-of-3 pivot and then a median-of-9 pivot for larger arrays with less or more than 40 items, respectively), `kmp` (Knuth-Morris-Pratt) scans a sequence while searching for an occurrence of a given subsequence, `bfs` (breadth-first search) searches in a graph in breadth-first order. We tested these programs instantiating them with respect to input lists or graph adjacency matrices that consist of 10, 50, 75 or 100 symbolic integers, respectively. For `kmp` , we fixed the length of the searched subsequence to three symbolic integers.

These sample programs are representative of algorithms for the traversal of sequential and recursive data structures. Traversals are key operations on data structures, and the traversal algorithms often characterize the performance of programs [21]. Moreover, these sample programs encompass worst case behaviours that manifest on program paths with both *regular* (in the case of `alternate_0`, `is_palindrome`, `merge_sort` and `bfs`) and *irregular* (in the case of `quicksort_jdk`, `kmp`, `memory_fill` and `dfsl`) decision sequences, and thus suite well for assessing, with a controlled and fair experiment, the novel characteristics of ESE with respect to the competing state-of-the-art techniques.

For each program and input size, we evaluated ESE both in absolute terms and in relative terms, with respect to the competing approaches at the state of the art. In absolute terms, we compared the execution cost measured by profiling the programs with the worst case inputs identified by ESE$_p y$, with the execution cost measured by using the manually identified worst case inputs. In relative terms, we compared the worst case inputs computed with ESE$_p y$ with the worst case inputs obtained with symbolic execution equipped with either (i) the classical depth-first (DFS), (ii) random path selection strategies (RAND) (iii) the guidance policies identified with either WISE [8] or (iv)

Table I. The programs considered in the experiments with their worst-case time complexity.

Program	Extended name	Category	Complexity
`alternate_0`	Alternate zeroes	Sequence checking	$\mathcal{O}(n)$
`is_palindrome`	Is palindrome?	Sequence checking	$\mathcal{O}(n)$
`merge_sort`	Merge sort	Sequence sorting	$\mathcal{O}(n \log n)$
`quicksort_jdk`	Quicksort (JDK implementation)	Sequence sorting	$\mathcal{O}(n^2)$
`kmp`	Knuth-Morris-Pratt	Sequence search	$\mathcal{O}(n)$
`memory_fill`	Memory fill	Sequence operation	$\mathcal{O}(n)$
`dfs`	Depth-first search	Graph search	$\mathcal{O}(n^2)$
`bfs`	Breadth-first search	Graph search	$\mathcal{O}(n^2)$

Table II. The main features and differences between ESE and the compared approaches.

Feature	ESE	WISE	SPF-WCA	EvoSuite
Program versions (**R**estricted / **U**nmodified)	U	R	R	U
Program path analysis (**E**xhaustive / **H**euristic)	H	E	E	H
Program path analysis (**S**ymbolic / **D**ynamic)	S	S	S	D

DOI: 10.1002/stvr

SPFWCA [9] approaches, respectively. We also compared ESE with the worst case inputs obtained with (v) EvoSuite adapted to use the count of executed instructions as fitness function.

Table II qualitatively classifies the features of ESE and the compared approaches, to enlighten their main differences. Both ESE and EvoSuite analyze unmodified versions of the target programs, while both WISE and SPF-WCA mostly refer to program versions in which they restrict the size of the inputs (first feature in the table). Thus, as we showed in Section 2, the latter approaches suffer from the limitation that they cannot find WCET test cases for the programs in which the WCET behaviour manifests only with inputs of at a given, non-small size, while the former approaches can in principle always identify the optimal WCET test cases for any target program. Both ESE and EvoSuite analyze an heuristically selected subset of the program paths, while both WISE and SPF-WCA exhaustively analyze all program paths under the chosen input restriction (second feature in the table). Thus, for programs in which the worst case executions regularly generalize across inputs of increasing size, WISE and SPF-WCA deterministically identify optimal WCET test cases, while the heuristic search procedures of ESE and EvoSuite cannot by-design guarantee the identification of optimal solutions. ESE shares with both WISE and SPF-WCA the characteristic of analyzing the target program with symbolic execution, while EvoSuite relies on purely dynamic analysis (third feature in the table). On the one hand, symbolic execution yields path condition formulas that precisely characterize in prepositional logic the classes of inputs that lead to executing different program paths, while EvoSuite computes only concrete inputs and analyze concrete executions and may thus sometimes be stuck in analyzing multiple different inputs that all execute the same program path. On the other hand, dynamic analysis is notoriously more efficient than static analysis, and thus EvoSuite will generally analyze many much more candidate solutions than ESE, when the two techniques are executed for equivalent time budgets.

To be fair, we implemented the four competing approaches that rely on symbolic execution, that is, DFS, RAND, WISE and SPF-WCA, on top of the same symbolic executor for Python that we use in ESE. DFS$_p y$ makes the symbolic executor visit the program paths in depth-first order. RAND$_p y$ works according to the random path selection strategy that we illustrated in Algorithm 3. WISE$_p y$ and SPF-WCA$_p y$ implement the algorithms of WISE and SPF-WCA, respectively, whose core step is to train a guidance policy for steering symbolic execution to identify the worst case (recall Section 2). WISE$_p y$ trains the guidance policy by analyzing the target program with initially unitary and then increasingly larger input bounds, until exhaustion of a training time budget. It then selects the worst path identified for the largest bound for which it successfully completes the exhaustive analysis of the program, and maps this path to a guidance policy, *decision point* $\longrightarrow$ *decisions*: the guidance policy associates the program decision points with the corresponding decisions (*none*, *true*, *false* or both *true* and *false*) taken at least once along that path. WISE$_p y$ trains the guidance policy in a similar way, but builds a finer-grained guidance policy specified as $\langle$*decision point*, *decision history*$\rangle \longrightarrow$ *decisions*, for the possible decision histories of a given length. SPF-WCA$_p y$ uses decision histories of Length 1.

We adapted EvoSuite by implementing the instruction-counting fitness function, using the instrumentation that EvoSuite natively includes to monitor the instructions traversed during the execution of the program under test, and adding the glue-code to make EvoSuite use this new fitness function. We ran EvoSuite with its default configuration and the new fitness function, plus the options `-generateTests` and `-Dchromosome_length=120`. The former option instructs EvoSuite to work with a population of test cases, rather than a population of test suites[§], while the latter option sets the maximum length of the generated test cases to a maximum of 120 lines of code. In particular, EvoSuite generates test cases that include the method calls to construct the lists (or the adjacency matrices) that the programs under test take as input, fill the lists (or the adjacency matrices) with values, and call the programs under test with these inputs. Thus, test cases that may consist of up to 120 lines of code provide sufficient room for EvoSuite to s et all values of the lists (or the adjacency matrices) considered in our experiments. Since EvoSuite works for programs in Java Bytecode, we

[§]We remark that using EvoSuite with a population of test suites, which is often the favoured way of executing EvoSuite in many scientific experiments with the tool [15], would be a bad choice in our context, where we look for a single optimal test case that maximizes the execution cost of the program under test.

DOI: 10.1002/stvr

implemented a Java equivalent version for all subject programs listed in Table I At the end of each experiment, we re-measured the WCET of the worst-case inputs generated with EvoSuite by executing these inputs against the Python version of the corresponding subject programs, aiming to obtain measurements that we can compare across the experiments with EvoSuite and ESE, respectively.

We ran ESE$_p y$, DFS$_p y$, RAND$_p y$ and EvoSuite against each program instance with a time budget of 60 min. For WISE$_p y$ and SPF-WCA$_p y$, we split the time budget in two tranches of 30 min each, for the training phase and the test generation phase, respectively, thus allowing for guidance policies trained on inputs of meaningful size, while still leaving adequate time to exhaustively analyze the decision points that the guidance policy does not constrain. Hereon, we omit the *'py'* subscript when referring to the prototypes.

4.3. RQ1—Sensitivity of ESE parameters

We studied the sensitivity of ESE with respect to the following key parameters of the algorithm:

- `popsize`, the number of individuals that are processed during each generation;
- `elitesize`, the amount of the best individuals that are propagated between subsequent generations;
- `mprob`, the frequency of mutation for new offsprings;
- `lsperiod`, the number of generations before a new phase of local search happens.

We studied the sensitivity of ESE by conducting experiments that test to what extent varying the values of these parameters impacts the precision of the WCET test cases that ESE computes, yielding test cases that approximate either well or badly the optimal WCET cost of the subject programs for large inputs of size 100. Specifically, we considered the following values of the parameters:

- `popsize` set to 10; 50; 75; 100;
- `elitesize` set to 0; 1; 5; 10;
- `mprob` set to 0.0; 0.2; 0.5; 1.0;
- `lsperiod` set to 0; 10; 50; 100.

In the spirit of pairwise testing, we selected a set of combinations of these values that suffice to evaluate the interactions between all values of any pair of parameters, that is, the 16 combinations indicated in the first four columns of Table III. The other columns of the table report the detailed results of the experiments with each subject program for each considered combination of parameter values.

Table III. Results of ESE configured with different values of its parameters.

popsize	elitesize	mprob	lsperiod	WCET test case execution cost (#instructions)							
				`alt.`	`is_p.`	`merg.`	`quic.`	`kmp`	`memo.`	`dfs`	`bfs`
10	0	0	0	549	73	4516	2808	584	396	141	194
10	5	0:5	10	601	202	4528	2665	596	422	193	222
10	1	0:2	50	591	202	4526	2730	592	422	193	220
10	10	1	100	589	202	4520	2635	588	418	191	220
50	5	0:2	0	559	202	4532	2256	598	404	163	198
50	0	1	10	595	202	4525	2767	598	418	193	218
50	10	0:5	50	595	202	4526	2647	595	421	193	220
50	1	0	100	567	202	4521	2682	590	413	193	216
75	0	0:5	0	571	75	4517	2849	600	405	145	198
75	5	0	10	601	202	4521	2711	589	422	193	222
75	1	1	50	587	202	4523	2578	611	421	193	222
75	10	0:2	100	575	202	4526	2742	594	415	191	222
100	5	1	0	563	202	4520	2503	593	404	159	204
100	0	0:2	10	601	202	4526	2776	587	422	193	222
100	10	0	50	585	202	4527	2664	597	420	193	222
100	1	0:5	100	585	202	4521	2638	592	411	193	210

ESE, Evolutionary Symbolic Execution; WCET, Worst-Case Execution Time.

DOI: 10.1002/stvr

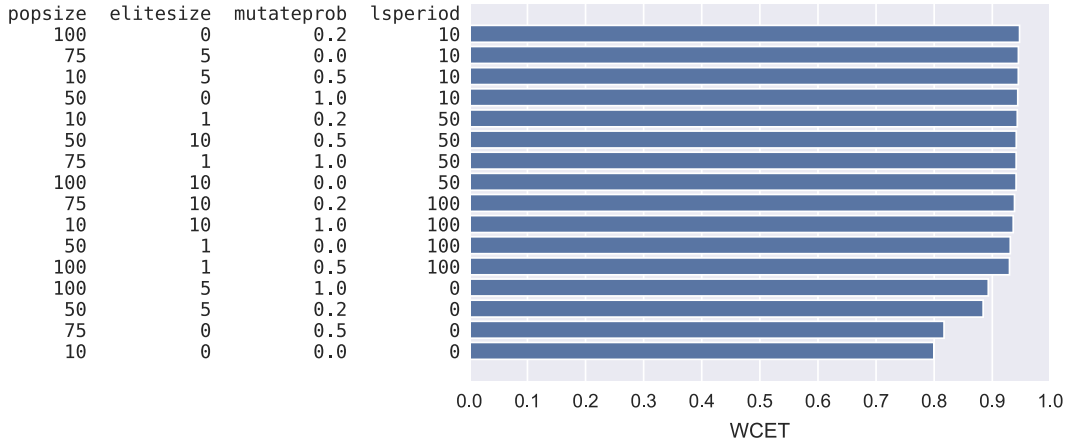

Figure 2. Comparison of the average (normalized) results of Evolutionary Symbolic Execution for the tested configurations of parameters. WCET, Worst-Case Execution Time

Figure 2 visualizes a summary of the results, comparing the WCET cost that we measured with the test cases computed with ESE, on average across the eight subject programs. To control for the different length of the worst-case paths of the subject programs, and avoid average figures dominated by the results obtained for the programs with the longest worst-case paths, we normalized the 'absolute' 'WCET' cost computed for each subject program with respect to the reference optimal value obtained when executing that program with the manually identified worst case input. The normalized values range between zero and one and express the distance from achieving WCET costs equal to the reference ones.

The data in Figure 2 suggest a low sensitivity of ESE with respect to the four parameters: 12 out of 16 considered combinations result in an average normalized WCET cost that ranges between 0.95 and 0.97. Looking at the distribution of the values of each single parameter across the tested combinations, the only clear observation that we can make is that the frequency of the local search phase influences the results: The rarer the frequency of the local search phase, the lower the rank of the corresponding experiments in Figure 2. Indeed, when the ESE algorithm is configured to execute without the local search phase (`lsperiod` $= 0$), ESE obtained the lowest WCET costs in these experiments.

In summary

ESE has a low sensitivity with respect to the parameters population size, elite size and mutation probability. However, it is very sensitive with respect to the choice of the period with which it executes the local search phase: too rare local search reduces the effectiveness of the technique.

Table IV. Parameters of ESE in the experiments.

Parameter	Description	Value
`popsize`	The size of the population	50
`elitesize`	The individuals retained as elite	5
`mprob`	The probability of mutation	0:2
`lsperiod`	The generations before local search	10
`lsattempts`	The changes during local search	25
`timeout`	The time budget for the search (min)	60

ESE, Evolutionary Symbolic Execution

DOI: 10.1002/stvr

Table V. Results obtained with ESE and the competing techniques for the subject programs for different size of the inputs.

Program	Size	WCET test case execution cost (#instructions)						
		Manual	ESE	RAND	DFS	WISE	SPF-WCA	EvoSuite
`alternate_0`	10	61	61	61	61	61	61	61
	50	301	301	277	269	267	301	301
	75	451	451	413	393	391	451	409
	100	601	601	543	517	517	601	547
`is_palindrome`	10	22	22	22	22	22	22	3
	50	102	102	37	102	102	102	3
	75	152	152	37	152	152	152	3
	100	202	202	37	202	202	202	3
`merge_sort`	10	279	279	279	277	276	279	279
	50	2019	2016	2003	1939	1939	2008	2018
	75	3249	3239	3218	3108	3108	3242	3249
	100	4549	4530	4501	4341	4341	4527	4535
`quicksort_jdk`	10	348	321	302	248	248	296	319
	50	1665	1485	1232	548	512	562	1407
	75	2528	2125	1750	673	635	685	2170
	100	3666	2730	2252	798	764	846	2696
`kmp`	10	83	83	80	83	83	83	54
	50	363	334	306	306	306	306	214
	75	538	478	444	442	442	442	317
	100	713	606	587	581	578	578	414
`memory_fill`	10	25	25	25	25	25	25	25
	50	222	222	212	188	186	186	222
	75	322	322	301	263	261	261	322
	100	422	422	390	336	334	334	422
`dfs`	10	18	18	18	18	18	18	18
	50	94	94	82	78	78	85	64
	75	156	156	122	116	116	116	86
	100	193	193	143	133	133	138	129
`bfs`	10	26	26	26	26	26	26	26
	50	114	114	102	114	114	114	114
	75	182	182	146	182	182	182	182
	100	222	222	176	222	222	222	222

DFS, classical depth-first; ESE, Evolutionary Symbolic Execution; RAND, random path selection strategies; SPF-WCA, Symbolic Path FinderWorst-Case Analysis; WCET, Worst-Case Execution Time; WISE, Worst-case Inputs from Symbolic Execution.

4.4. RQ2—Effectiveness of ESE

Based on the sensitivity analysis that we discussed above, we selected a balanced configuration of ESE for studying the other research questions, by choosing, for each of the four parameters in Figure 2, the value that occurs more times in the top four configurations in the figure. For the parameters `popsize` and `mprob`, for which each values occurs exactly one time, we arbitrarily selected the second value of the corresponding scale. Table IV summarizes the resulting configuration.

Table V reports the results of our experiments with the selected configuration. For each program listed in column 'program' and each input size listed in column 'size', the seven columns below 'WCET test case execution cost' report the execution cost, expressed as the number of executed instructions, which we obtained by profiling the program with the worst case inputs identified either manually (column 'manual') or with the techniques ESE, RAND, DFS, WISE, SPF-WCA and Evo-

DOI: 10.1002/stvr

Suite, respectively. For the experiments with the techniques ESE, RAND and EvoSuite, the data are the average values across 10 runs, to control for the randomness in these techniques.

Investigating RQ2, we study the effectiveness and the significance ESE by comparing it with the baseline strategies RAND and DFS. For inputs bounded at size 50, 75 and 100, ESE consistently outperforms the baseline random strategy RAND, computes the same optimal WCET test case as DFS for the programs `is_palindrome` and `bfs` and consistently outperforms DFS for all other programs. The dominance of ESE on RAND and DFS confirms that ESE successfully steers the search towards the worst-case program path and successfully contrasts the path-explosion issues incurred with the systematic strategy of DFS.

In summary

ESE successfully steers symbolic execution towards generating WCET test cases, significantly outperforming both the baseline strategies of symbolically executing program paths in either depth-first or random order.

4.5. RQ3—Comparison with WISE and SPF-WCA

With reference to RQ3, we compare ESE with the state-of-the-art techniques WISE and SPF-WCA. Below, we elaborate on the comparison by restricting our attention to ESE and SPF-WCA, since WISE is always equivalent to or worse than SPF-WCA in our experiments. The data in Table V reveal the mutually complementary strengths of these techniques. ESE outperforms SPF-WCA in the experiments with `quicksort_jdk`, `kmp`, `memory_fill` and `dfs`, is comparable to (even though slightly better than) SPF-WCA for `merge_sort` and is equivalent to SPF-WCA for `alternate_0`, `is_palindrome` and `bfs`.

The experiments in which ESE outperforms SPF-WCA confirm the observations that we made in Section 2 about (i) programs for which SPF-WCA cannot compute a conclusive guidance policy (e.g., `dfs` and `kmp`) and thus falls back to the same effectiveness of DFS, and (ii) programs for which the worst-case behaviour observed on small inputs does not generalize for large inputs (e.g. `memory_fill` and `quicksort_jdk`).

For these programmes, SPF-WCA ends up with analyzing sets of programme paths that may not include the worst-case path, while ESE suites better by sampling the target program scope directly.

For example, in the case of `quicksort_jdk`, the in-depth analysis of the bad results of SPFWCA reveals that it is the expected consequence of the change of behaviour of the programme with small or large arrays, respectively: The behaviour of the programme that WISE and SPF-WCA observe in 30 min with arrays up to a maximum of seven items does not generalize when executing the programme with arrays of size 10, 50, 75 and 100.

On the other side of the spectrum, `merge_sort` has an almost regular worst case, which mostly consists of alternating decisions when merging the items of the sorted sublists. This type of worst-case behaviour is similar to the one that we described in Section 2 with reference to the programme `alternate_0` and can be (at least partially) captured with a guidance policy in the same way as SPF-WCA.

In general, both WISE and SPF-WCA assume the existence (and the availability) of a monotone relation between inputs of increasing size and increasing worst-case execution costs. For the programmes considered in our experiments, which take inputs shaped as list and graph structures, it is natural to identify this relation based on the size of the data structures, for example, devising a guidance policy for `merge_sort` by considering the sorting of lists of small size and then using that guidance policy to analyze the worst-case execution cost when sorting large lists. However, for many practical programmes, which may take multiple inputs, shaped as various types of interwoven data structures, satisfying this assumption may not be easy: Each different input may or not participate in the worst-case behaviour of the programme, and the definition of the size increment related

DOI: 10.1002/stvr

to the combination of the participating inputs can be peculiar. ESE dismisses this requirement by working directly on the target programme scope and can thus natively address this general case.

In summary

As expected, ESE outperforms WISE and SPF-WCA in programmes for which these techniques cannot compute conclusive guidance policies, and those for which the worst-case behaviour observed on small inputs does not generalize for large inputs.

4.6. RQ4 – Comparison with EvoSuite

We elaborate on research question RQ4 based on the comparison of the results of ESE and EvoSuite in Table V. In our experiments, ESE outperforms EvoSuit for `alternate_0`, `is_palindrome`, `kmp` and `dfs`, is equivalent to EvoSuit for `memory_fill` and `bfs`, and is comparable to (though slightly worse) EvoSuit for `merge_sort` and `quicksort_jdk`.

EvoSuit yields the poorest results with `is_palindrome` and `kmp`, where it does not identify the optimal WCET test case in any experiment, not even in the experiments with inputs of Size 10. The key observation in these experiments is that, for both these programmes, the worst-case programme path depends on equality relations between distinct values of the respective input lists, and it is thus very hard for EvoSuite to identify satisfying inputs by independently sampling the values of the list items. These experiments provide evidence of the benefits of EvoSuite that explicitly represents the relevant equalities in the path conditions of the candidate solutions, confirming that EvoSuite is a better suited technique than purely dynamic search based testing techniques for a problem like WCET testing, for which the solution is monotonic in the length of the programme paths. When (as in the case of paths the depend on strict equality conditions) the possible input values unevenly distribute across the classes of inputs that trigger distinct programme paths, it can be sometimes hard to cover programme paths that are elicited by classes of inputs with low cardinality, for example, like in `is_palindrome` where most input values lead to suboptimal programme paths and very few specific inputs trigger the worst-case programme paths. Instead, the symbolically computed path conditions specifically identify the distinct feasible programme paths, thus increasing the probability of sampling the relevant ones.

EvoSuite works slightly better with `alternate_0` and `dfs` and has optimal performance (like ESE) with `memory_fill` and `bfs`. In these programmes, the worst-case values of the input list items are mutually independent. This simplifies the problem to be solved with EvoSuite. Nonetheless, when we increase the size of the problems in `alternate_0` and `dfs`, the path condition-level optimizations of ESE suite better than the input-level optimizations of EvoSuite for identifying solutions to the strict equality constraints in these two programmes.

For `merge_sort` and `quicksort_jdk`, EvoSuite and ESE have very close performance, with ESE performing slightly better than EvoSuite for `quicksort_jdk`, and EvoSuite performing slightly better than ESE for `merge_sort` in the experiments with largest inputs. Since both these programmes implement sorting algorithms, their programme paths depend on comparing the relative order of the inputs, but not on strict equality comparisons. Thus, on one hand, the experiments with these two programmes indicate that the high efficiency of the purely dynamic analysis of EvoSuite pays well if the programme under test has limited dependencies on equality comparisons. On the other hand, these results further strengthen the observation that we made in the experiments with the other programmes, pinpointing in the handling of the equality comparisons the distinctive strength of the path condition-level evolutionary operators of ESE: By reasoning on the path conditions of the programme paths, the evolutionary operators of ESE result in effectively sampling the classes of inputs (represented as path conditions) that trigger the execution of distinct programme paths, more effectively than directly sampling the possible input values as in EvoSuite.

In summary

The comparison between the experimental results that we achieved with ESE and the version of EvoSuite that we adapted with the WCET fitness function indicate that the path condition-level

DOI: 10.1002/stvr

evolutionary operators of ESE significantly contribute to the effectiveness of the ESE approach, beyond just the merit of the WCET fitness function.

4.7. Threats and limitations

The goal of these experiments was primarily to provide empirical evidence of the implications of the regular and irregular worst case phenomena in ESE and the competing techniques, but as the main threat, we cannot claim that the considered programmes are representative of all the challenges that may emerge when applying our technique to general purpose industrial scale programmes, and thus our result may not generalize. A larger selection of benchmarks would be necessary to systematically assess the absolute and mutual benefits of ESE, WISE and SPF-WCA. Moreover, we need to further investigate to which extent the theoretical and practical limitations of symbolic execution impact ESE, for example, constraints that the constraint solver cannot cope with, low-level operations that can be hard to simulate symbolically and the possible mismatch between the static counting of the traversed instructions and the actual execution cost of the programmes.

5. RELATED WORK

Many static WCET analyzers for hard real-time systems combine symbolic execution with timed automata and static analysis to compute WCET estimations [1–7]. In the context of these techniques, symbolic execution complements (mostly static) analysis procedures, assisting the pruning of infeasible execution paths that would jeopardize the precision of the estimations. Knoop et al. augment the r-TuBound WCET analyzer by symbolically executing the identified worst path to determine whether the path is infeasible; if so, they iterate the analyzer with refined constraints to find a different path [6]. Biere et al. use symbolic execution to enhance the ability of the analyzer to compute loop bounds [5]. Kebbal end Sainrat rely on symbolic execution to automatically extract information about the programme semantics that they used to tighten the WCET estimates [3]. In general, these techniques pursue the goal of computing WCET estimations, but not the generation of WCET test cases, and can hardly cope with general purpose programmes with large path spaces.

Our work is closely related to techniques that rely on symbolic execution to generate test data for worst case and load testing, security exploit synthesis and structural code coverage [8, 9, 12, 22–27]. We have extensively commented on state of the art in WCET testing [8, 9] in Section 2, and experimentally compared the existing techniques, WISE and Symbolic Path FinderWorst-Case Analysis (SPF-WCA), with our approach in Section 4. Zhang et al., propose a technique for load testing that iterates symbolic execution to explore increasingly longer programme paths, with a greedy path selection heuristic that, at each iteration, continues to explore only the paths that yielded the highest workload at the previous iteration [22]. Avgerinos et. al. use symbolic execution to find working exploits for security vulnerabilities, using path selection heuristics based on domain knowledge on the vulnerabilities [23]. In general, many symbolic-execution-based test generators embed path selection heuristics that address structural code coverage [12, 16, 24–27]. The approach discussed in this paper is the first to investigate an evolutive path selection heuristic for symbolic execution, where a cost model guides the search. In the future, we aim to study if ESE can be beneficial for other goals other than WCET testing.

The area of Search-Based Software Testing (SBST) encompasses techniques that use search-based optimization algorithms for the generation of test data [14]. So far, a substantial research effort in the area has targeted structural coverage [15, 28–35]. In particular, we briefly survey the techniques that investigated combinations of SBST and symbolic execution [16, 36, 37]. Other researchers exploit symbolic execution to optimize the evolutionary algorithms used in SBST [38–41]. Xie et. al. and Baluda investigate search-based path selection strategies to steer symbolic execution towards paths with higher chances to execute uncovered branches [42, 43]. None of these previous SBST techniques targeted WCET testing so far, and none of them investigated a search strategy based on combining the path conditions of the incrementally analyzed programme paths, as we propose in this paper.

DOI: 10.1002/stvr

6. CONCLUSIONS

In this paper, we presented a novel technique for WCET testing of software programmes. We provided compelling examples of the limitations of the state-of-the-art techniques based on guided symbolic execution and proposed a novel technique that combines symbolic execution with an evolutionary algorithm that steers the incremental analysis of increasingly worse programme paths.

The experiments presented in this paper support our hypothesis that there exists a class of programmes for which WCET testing cannot be addressed with the guided symbolic execution approach and indicate that our ESE technique provides a viable solution for WCET testing of these programmes. The experiments on the sensitivity of ESE confirmed the importance the local search phase of our algorithm, and the comparison with EvoSuite indicated the effectiveness of the evolutionary operators that ESE defines based on symbolic execution with respect to traditional operators that manipulate concrete inputs.

We believe that the benefits of ESE over guided symbolic execution generalize to practical programmes, which may take multiple inputs, shaped as various complex data structures, each impacting in its own specific way on the worst-case behaviour of the programme. ESE may conceptually address such programmes directly, while guided symbolic execution cannot. Currently, we are porting ESE on a mature symbolic execution framework, that is, JBSE [44], to enable experiments with ESE on more general programmes than the ones considered in this paper.

ACKNOWLEDGMENTS

This work has been partially supported by the GAUSS research project, funded by Ministero dell'Istruzione, dell'Università e della Ricerca (MIUR) under the PRIN 2015 programme (Contract 2015KWREMX).

REFERENCES

1. Lundqvist T., Stenström P. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems* 1999; **17**(2-3):183–207.
2. Stappert F., Altenbernd P. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture* 2000; **46**(4):339–355.
3. Kebbal D., Sainrat P. Combining symbolic execution and path enumeration in worst-case execution time analysis. In *International Workshop on Worst-Case Execution Time Analysis (WCET)*, Mueller F (ed.), 2006.
4. Benhamamouch B., Monsuez B., Védrine F. Computing WCET using symbolic execution. *International Conference on Verification and Evaluation of Computer and Communication Systems (VECoS)* 2008:128–139.
5. Biere A., Knoop J., Kovács L, Zwirchmayr J. The auspicious couple: symbolic execution and WCET analysis. *International Workshop on Worst-Case Execution Time Analysis (WCET)* 2013:53–63.
6. Knoop J., Kovács L, Zwirchmayr J. WCET Squeezing: on-Demand feasibility refinement for proven precise WCET-bounds. *International Conference on Real-Time Networks and Systems (RTNS)* 2013:161–170.
7. Luckow K. S., Thomsen B. Symbolic execution and timed automata model checking for timing analysis of Java Real-Time Systems. *EURASIP Journal on Embedded Systems* 2015; **2015**(1):2.
8. Burnim J., Juvekar S., Sen K. WISE automated test generation for worst-case complexity. *IEEE/ACM International Conference on Software Engineering (ICSE)* 2009:463–473.
9. Luckow K., Kersten R., Pasareanu C. Symbolic complexity analysis using context-preserving histories. *IEEE International Conference on Software Testing, Verification and Validation (ICST)* 2017:58–68.
10. King J. C. Symbolic execution and program testing. *Communications of the ACM* 1976; **19**(7):385–394.
11. Clarke L. A. A program testing system. *ACM Annual Conference* 1976:488–491.
12. Cadar C., Sen K. Symbolic execution for software testing: three decades later. *Communications of the ACM* 2013; **56**(2):82–90.
13. Roper M. Computer aided software testing using genetic algorithms. *International Software Quality Week (QW)* 1997.
14. McMinn P. Search-based software test data generation a survey. *Software Testing, Verification and Reliability* 2004; **14**(2):105–156.
15. Fraser G., Arcuri A. Whole test suite generation. *IEEE Transactions on Software Engineering* 2013; **39**(2):276–291.
16. Braione P., Denaro G., Mattavelli A., Pezzè M. Combining symbolic execution and search-based testing for programs with complex heap inputs. *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* 2017:90–101.

DOI: 10.1002/stvr

17. Panichella A., Kifetew F.M., Tonella P. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 2018; **44**(2):122–158.
18. Aquino A., Denaro G., Salza P. Worst-case execution time testing via evolutionary symbolic execution. *International Symposium on Software Reliability Engineering (ISSRE)* 2018:76–87.
19. De Moura L., Bjørner N. Z3: an efficient SMT solver. *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)* 2008:337–340.
20. Saxena P., Akhawe D., Hanna S., Mao F., McCamant S., Song D. A symbolic execution framework for JavaScript. *IEEE Symposium on Security and Privacy (SP)* 2010:513–528.
21. Padhye R., Sen K. Travioli: a dynamic analysis for detecting data-structure traversals. *IEEE/ACM International Conference on Software Engineering (ICSE)* 2017:473–483.
22. Zhang P., Elbaum S., Dwyer M. B. Automatic generation of load tests. *IEEE/ ACM International Conference on Automated Software Engineering (ASE)* 2011:43–52.
23. Avgerinos T., Cha S.K., Rebert A., Schwartz E.J., Woo M., Brumley D. Automatic exploit generation. *Communications of the ACM* 2014; **57**(2):74–84.
24. Tillmann N., de Halleux J. Pex: white box test generation for.NET. *International Conference on Tests and Proofs (TAP)* 2008:134–153.
25. Cadar C., Dunbar D., Engler D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. *Symposium on Operating Systems Design and Implementation (OSDI)* 2008:209–224.
26. Burnim J., Sen K. Heuristics for scalable dynamic test generation. *IEEE/ACM International Conference on Automated Software Engineering (ASE)* 2008:443–446.
27. Braione P., Denaro G., Mattavelli A., Vivanti M., Muhammad A. Software testing with code-based test generators: data and lessons learned from a case study with an industrial software component. *Software Quality Journal* 2014; **22**(2):311–333.
28. Fraser G., Arcuri A., McMinn P. A memetic algorithm for whole test suite generation. *Journal of Systems and Software* 2015; **103**(C):311–327.
29. Wegener J., Baresel A., Sthamer H. Evolutionary test environment for automatic structural testing. *Information and Software Technology* 2001; **43**(14):841–854.
30. Fraser G., Zeller A. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering* 2012; **38**(2):278–292.
31. Vivanti M., Mis A., Gorla A., Fraser G. Search-based data-flow test generation. *IEEE International Symposium on Software Reliability Engineering (ISSRE)* 2013:370–379.
32. Tonella P. Evolutionary testing of classes. *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* 2004:119–128.
33. Walcott K.R., Soffa M.L., Kapfhammer G.M., Roos R. S. Timeaware test suite prioritization. *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* 2006:1–12.
34. Colanzi T.E., Assunção W K G, Vergilio S.R., Pozo A. Integration test of classes and aspects with a multi-evolutionary and coupling-based approach. *International Symposium on Search Based Software Engineering (SSBSE)* 2011:188–203.
35. Gross F., Fraser G., Zeller A. Search-based system testing: high coverage, no false alarms. *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* 2012:67–77.
36. Xie T., Marinov D., Schulte W., Notkin D. Symstra: a framework for generating object-oriented unit tests using symbolic execution. *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)* 2005:365–381.
37. Inkumsah K., Xie T. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. *IEEE/ACM International Conference on Automated Software Engineering (ASE)* 2008:297–306.
38. Baars A., Harman M., Hassoun Y., Lakhotia K., McMinn P., Tonella P., Vos T. Symbolic search-based testing. *IEEE/ACM International Conference on Automated Software Engineering (ASE)* 2011:53–62.
39. Galeotti J.P., Fraser G., Arcuri A. Improving search-based test suite generation with dynamic symbolic execution: IEEE International Symposium on Software Reliability Engineering (ISSRE), 2013.
40. Malburg J., Fraser G. Combining search-based and constraint-based testing. *IEEE/ACM International Conference on Automated Software Engineering (ASE)* 2011:436–439.
41. Lakhotia K., Harman M., McMinn P. Handling dynamic data structures in search based testing. *Genetic and Evolutionary Computation Conference (GECCO)* 2008:1759–1766.
42. Xie T., Tillmann N., de Halleux P., Schulte W. Fitness-guided path exploration in dynamic symbolic execution. *International Conference on Dependable Systems and Networks (DSN)* 2009:359–368.
43. Baluda M. EvoSE: evolutionary symbolic execution. *International Workshop on Automating Test Case Design, Selection and Evaluation (A-TEST)* 2015:16–19.
44. Braione P., Denaro G., Pezzè M. JBSE A symbolic executor for Java programs with complex heap input. *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)* 2016:1018–1022.

DOI: 10.1002/stvr