Master Thesis

WhiteSyn

A fully automatic test generation tool for performance bugs

Tim Linggi

Dr. Luca Della Toffola, Michael Faes
Responsible assistants

Prof. Thomas R. Gross
Laboratory for Software Technology
ETH Zurich

November 28, 2018

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

LST
Laboratory for Software Technology

Abstract

Unlike functional bugs, performance bugs are generally not well covered by unit tests in popular software projects [14], even tough their presence may drastically decrease the performance of an application. Usually, only a small group of developers maintain performance tests and they are rarely updated. While tools for discovering potential performance issues exist, many require manual work by a developer (e.g. writing tests or inspecting code), which may not be desirable.

This report presents an automatic test generation approach that uses static control-flow graph analysis and a genetic algorithm. We have implemented this approach with the prototype tool WhiteSyn, a fully automatic performance test generation tool. WhiteSyn is able to generate tests with large input size, which is usually a prerequisite for performance bugs to manifest. With a static analysis, WhiteSyn finds desired paths through a system under test and uses a genetic algorithm to generate unit tests that trigger these paths and iterate desired loops multiple times. As a proof of concept, WhiteSyn is able to generate tests that expose redundant computations, a common class of performance bugs. Testing on known performance issues has shown that WhiteSyn is able to generate tests with large input size and tests that expose previously known redundant computations.

Contents

1 Introduction

Performance bugs[1] are ubiquitous [11] [15]. However, many performance bugs remain undetected, since they are only exposed by a specific input or do not cause a noticeable runtime increase, even if triggered. When compared to functional bugs, usually less attention is paid to assessing the quality of a project in terms of performance [14]. Additionally, determining what constitutes a performance bug is not as straight-forward as comparing the actual outcome to an expected outcome, since a clear definition of the expected performance may not available or not even possible [17].

This report has two key contributions. First, we formulate a white box test generation approach for performance bugs based on static CFG analysis and a genetic algorithm. The genetic algorithm takes a desired path through the CFG and the number of required loop iterations into account. Second, we implement this approach as the prototype tool WhiteSyn. Throughout this report, we will focus on WhiteSyn, as it is the prototype for the first key contribution, and discuss the advantages and drawbacks of the approach in general in the evaluation and the conclusion.

WhiteSyn is a fully automatic test generation tool that analyzes a system under test (SUT) to generate performance tests with large input size, which is a common prerequisite for performance issues to manifest. As a proof of concept, WhiteSyn is able to find redundant computations in nested loops, which are of particular interest, since their impact on performance is amplified due to the nesting.

WhiteSyn is inspired by PerfSyn [27], which is a test generation tool as well. In contrast to PerfSyn however, WhiteSyn does a white box analysis instead of a black box analysis and a genetic algorithm [30] is used to generate tests, rather than a combinatorial search. Finding redundancies is achieved with the Toddler oracle [18]. Previous work on performance test generation deals with worst-case complexity [24] [5], requires manual effort [18] [21] or relies on predefined patterns [7]. WhiteSyn on the other hand, bases its analysis on loop iterations, requires no input other than the SUT and is fully automatic. Static CFG analysis and a genetic algorithm to generate tests was proposed in [23]. This concept was later applied to generate unit tests by [28]. However, both of these approaches do not focus on performance tests and do not take the number of loop iterations into account.

Initially, WhiteSyn statically finds control-flow paths through the SUT that expose interesting behavior, e.g. reach a certain loop depth. Then, a genetic algorithm is used to generate tests that trigger these paths. Given such a test and the SUT instrumented according to a redundancy oracle, executing the generated test can show performance problems, such as redundant computations, that

[1] "A bug that affects speed or responsiveness" as defined by Bugzilla (`https://bugzilla.mozilla.org/describekeywords.cgi`)

WhiteSyn reports to the user.

1.1 Problem Definition

WhiteSyn aims to *fully automatically* find potential performance bugs and generate one or many *tests* (or *input programs*) that may expose these bugs. Since performance bugs usually manifest only with a certain input size (e.g. a large list as argument of a method under test), one goal of WhiteSyn is to generate tests that achieve this input size and therefore trigger long running executions. Throughout this report, we will refer to "long running" in terms of the number of loop iterations. Therefore, a long running test for a SUT iterates some loops at least a certain number[2] of times.

As a proof of concept, WhiteSyn uses the generated, long running tests to find redundant computations (or simply *redundancies*), which are a common class of performance bugs. WhiteSyn aims to expose redundancies in nested loops in particular, since their impact on performance is larger due to the amplification caused by the nesting.

To summarize, the two main goals of WhiteSyn are:

G1 Generate tests with large input size for a SUT that trigger long running, non-crashing executions.

G2 Expose redundancies in a SUT and provide tests that trigger them.

1.2 Motivating Example

To illustrate WhiteSyn, we provide a motivating example shown in Listing 1.1. Method `subtract` subtracts two collections from each other, i.e. the resulting collection contains all elements that are in `col1` but not in `col2`. For this purpose, a loop iterates through `col1` and checks for each element, if it is in `col2` as well. If not, it appends the element to a list that is later returned. However, `col2` might be a collection whose `contains` has linear complexity (e.g. an `ArrayList`) as shown on `Line 10` in Listing 1.2. In this case, `subtract` has super-linear complexity in regard to the size of `col1`, which is not desirable. On the other hand, when calling `subtract` with a `HashSet` as second parameter, as shown on `Line 13`, `subtract` has linear complexity.

Therefore, `subtract` contains a possibly redundant inner loop, i.e. a performance bug, that is only triggered when the second argument's `contains` method has linear complexity and the two collections are disjoint. `Line 5` in Listing 1.1 shows how to fix the performance bug: one can wrap `col2` into a `Set` whose `contains` has constant complexity.

Let us compare the two different `contains` methods called by `Line 10` and `Line 13` in Listing 1.2, respectively. Listing 1.3 shows an implementation of `contains` of an array-based list. Given an element to search for, it iterates through all the elements in the list and therefore has linear

[2]This number is discussed later

Listing 1.1: A motivating example containing a performance bug.

```
class Utils {
   Collection subtract(Collection col1, Collection col2) {
      Collection result = new ArrayList();
      // Bug fix:
      // col2 = new HashSet(col2);
      for (Object el : col1) {
         //This 'contains' call might have linear complexity
         if (!col2.contains(el)) {
            result.add(el);
         }
      }
      return result;
   }
}
```

Listing 1.2: An illustration of how the method from Listing 1.1 can be used.

```
void redundance() {
   Collection list1 = new ArrayList();
   Collection list2 = new ArrayList();
   Collection set1 = new HashSet();

   //Fill both lists and the set
   ...

   //This call has super-linear complexity
   Utils.subtract(list1, list2);

   //This call has linear comlexity
   Utils.subtract(list1, set1);
}
```

Listing 1.3: Method `contains` of an array-based list.

```
class List {
   Object[] elements;
   ...
   boolean contains(Object element) {
      for (Object e : elements) {
         if (e.equals(element)) {
            return true;
         }
      }
      return false;
   }
}
```

Listing 1.4: Method `contains` of a hash-based set. For simplicity, we assume that there are no hash collisions.

```
class Set {
   boolean[] contained;
   ...
   boolean contains(Object element) {
      int index = hash(element);
      return contained[index];
   }
}
```

complexity. On the other hand, Listing 1.4 illustrates `contains` of a hash-based set, which hashes the element to an index and checks if the flag at this index in a boolean array is set. If we assume that there are no hash collisions, this does not require a loop[3].

According to ***G2***, the performance bug shown in Listing 1.1 should be found and a test that triggers the performance bug should be generated. Given only the target method's signature, WhiteSyn is able to fully automatically generate such a test, which can be found in Listing 1.5. The test creates two lists and fills them with random strings, such that the lists are *disjoint*. It is important that the lists are disjoint, otherwise the loop in the list's `contains` method would not iterate through the entire list and not trigger the performance bug. The reason both lists are filled with 7 elements is discussed in Section 4.2.

[3]We make this assumption for illustration purposes, however, it does not hold for all hash sets.

Listing 1.5: A fully automatically generated test that triggers the performance bug from Listing 1.1.

```
void test0() {
   LinkedList<String> linkedList0 = new LinkedList<String>();
   ArrayList<String> arrayList0 = new ArrayList<String>();
   arrayList0.add("cvePbe2");
   arrayList0.add("cvePbe2");
   arrayList0.add("cvePbe2");
   arrayList0.add("cvePbe2");
   arrayList0.add("cvePbe2");
   arrayList0.add("cvePbe2");
   arrayList0.add("cvePbe2");
   linkedList0.push("Q&");
   linkedList0.push("Q&");
   linkedList0.push("Q&");
   linkedList0.push("Q&");
   linkedList0.push("Q&");
   linkedList0.push("Q&");
   linkedList0.push("Q&");
   Collection<String> collection0 = subtract(arrayList0, linkedList0);
}
```

2 Background

2.1 Control-Flow Graphs

A *control-flow graph* (*CFG*) [1] is a directed graph that represents the control flow of a single method. Its components are *basic blocks* (or *nodes*) and *transitions* (or *edges*). A basic block is a sequence of code without any jumps into or out of it, except for its entry and exit. Each CFG has a single start block (the method entry) and possibly multiple end blocks, one for each `return` and uncaught `throw` statement of the method[1]. Transitions represent *possible* control flow through the CFG. However, not every transition might be feasible, but each feasible control flow must be represented by the CFG.

An illustration of the CFG of the motivating example from Listing 1.1 can be found in Figure 2.1a. The two different implementations of `contains` from Listing 1.3 and Listing 1.4 are illustrated in Figure 2.1b and Figure 2.1c, respectively.

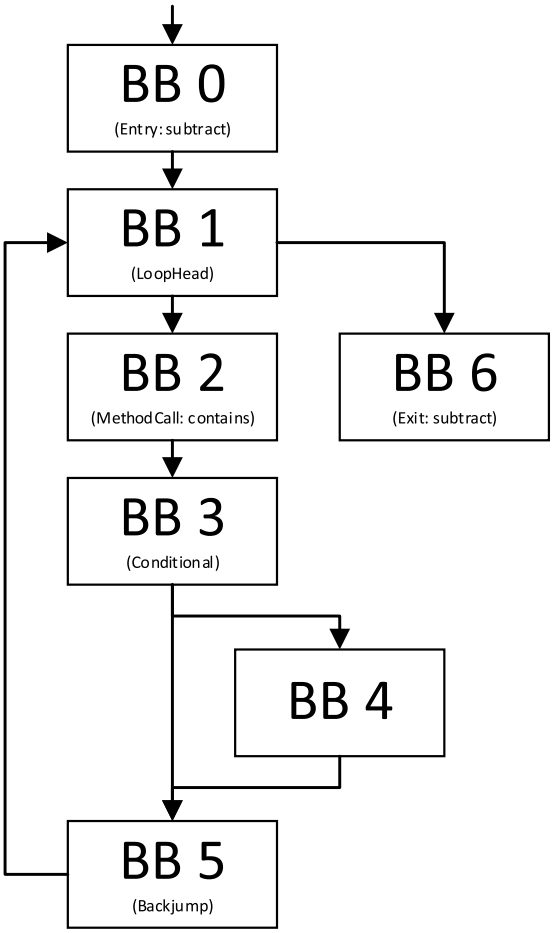
2.2 Genetic Algorithms

A *genetic algorithms* (GA) [30] is a heuristic used to solve search and optimization problems, which is inspired by the process of natural selection. A *population* of *individuals* (also called *candidates* or *candidate solutions*) is *evolved* by selecting fitter individuals in each step. To determine the *fitness* of an individual, a *fitness function* is used (see Section 2.3). The fitness of an individual is expressed as a value in the range $[0, 1]$, with a fitness of 0 being optimal. The goal of the GA is to find one or more *solutions* to a problem that is defined by a fitness function (also called *covering a goal*). A GA might aim to find solutions for multiple problems at once. In this case, each problem is defined by its own fitness function.

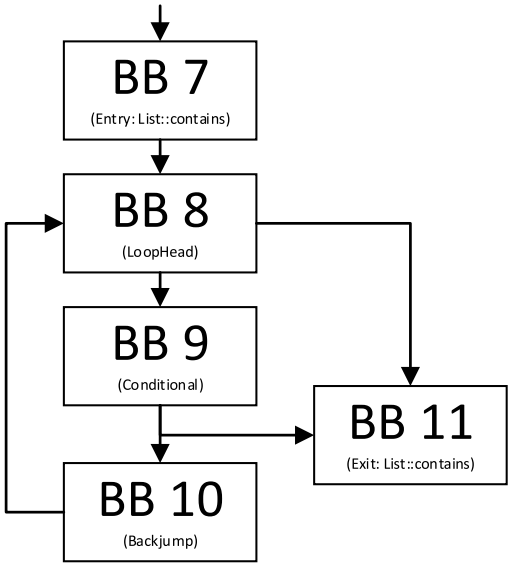
A GA is an iterative algorithm that starts with an initial (usually random) population of individuals and evolves them until either all goals are covered or a timeout occurs. A goal is considered to be covered, if for at least one individual in the population, the fitness function that corresponds to the goal evaluates to 0.

During each evolution step, various operators can be applied to create the new *generation* of individuals. The most commonly used operators are *selection*, *crossover* and *mutation*. During selection, fitter individuals are more likely to be chosen to be in the next generation. These fitter individuals are then crossed over with other individuals, e.g. by exchanging parts of the individuals.

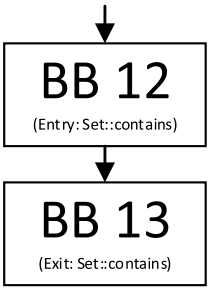
[1] An artificial end block might be added, such that each CFG has a single exit.



(a) The CFG of the `subtract` method from Listing 1.1.



(b) The CFG of the List's `contains` method from Listing 1.3.



(c) The CFG of the Set's `contains` method from Listing 1.4.

Figure 2.1: The CFGs of the methods involved in the motivating example.

Lastly they might be mutated, usually at random. An illustration of the control flow of a GA can be found in Figure 2.2.

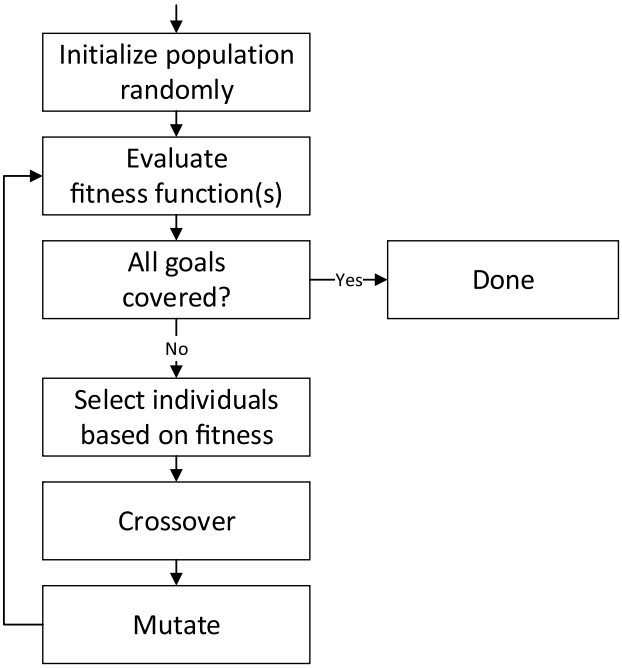


Figure 2.2: The control flow of a genetic algorithm.

For our context, let us consider the problem of finding a program that follows a predetermined path p through the SUT. Such a path consists of a list of basic blocks. An individual is a program, represented as a list of statements, and the control flow of a solution program follows p, i.e. visits all blocks that are in p. The fitness function takes a program as input and maps it to a ratio of unvisited blocks of p to total blocks of p. If the fitness is 0, all blocks of p are visited. During evolution, programs are mutated by adding, removing and changing statements. For crossover, a random index in the statement lists of two programs is chosen and the statements after these points are swapped. The more blocks of p a program covers, the more likely it is to be selected for the next generation.

2.3 Types of Fitness Functions

Let us define $\mathcal{I}$ to be the set of all individuals. A single-objective fitness function for a single individual is therefore described as

$$f_s : \mathcal{I} \to [0, 1] \tag{2.3.1}$$

One approach that uses a single-objective fitness function is called *whole suite* (WS) [9]. In this approach, the individuals are not programs, but rather test suites (sets) of programs (later referred to as $T = [t_1, ..., t_k]$ with t_i being a single program in the suite). Therefore, a solution to an optimization problem may be a set of programs instead of only a single program. Note that the name *single-objective* refers to the single fitness value calculated for an individual by this approach.

It is common to target multiple goals at once with the WS approach, but the GA still aims to minimize a single fitness value. Given the set of all single fitness functions $\mathcal{F}$, the number of goals n and the set of all test suites $\mathcal{I}_{ws}$, the WS fitness function is described as

$$f_{ws} : \mathcal{F}^n \times \mathcal{I}_{ws} \to [0, 1] \tag{2.3.2}$$

Algorithm 1 illustrates an implementation of the WS approach. The population is evolved for multiple goals at once, each defined by its own fitness function $f_i \in \mathcal{F}$. The fitness of the suite is calculated as a normalized sum of the best evaluations of all f_i. We therefore can reuse generated programs within a suite to cover multiple goals.

However, by summing up the single fitnesses to one single value, information about the individual goals is lost. Therefore, the fitness of the whole suite has limited expensiveness in terms of fitnesses of individual goals. This manifests in particular when dealing with goals that are not closely related or even mutually exclusive, because improving the single fitness for one goal might worsen the single fitness for another goal, without this trade-off being reflected by the fitness of the suite. For example, if four uncovered goals evaluate to 0.25 fitness each, the suite has the same fitness (0.25) as a suite with three covered goals (fitness 0.0) and one infeasible goal (fitness 1.0).

Algorithm 1: The fitness function of the WS approach.

Name: getWholeSuiteFitness
Input: Fitness functions $F = [f_1, ..., f_n]$ for goals to cover, candidate solution (test suite) $T = [t_1, ..., t_k]$
Output: Fitness $\phi \in [0, 1]$ of the suite candidate solution
$\phi \leftarrow 0.0$;
foreach $i \in [1, n]$ **do**
  $\phi_i \leftarrow \min_j f_i(t_j)$; `// Get fitness for best test j for goal i`
  $\phi \leftarrow \phi + \phi_i$; `// Sum up fitnesses of all goals`
end
$\phi \leftarrow \phi / n$; `// Normalize`
return ϕ

Algorithm 2: The fitness function of the MOSA approach.

Name: getMOSAFitness
Input: Fitness functions $F = [f_1, ..., f_n]$ for goals to cover, candidate solution t
Output: Fitness vector $\phi \in [0, 1]^n$ of test candidate solution
$\phi \leftarrow$ initialize vector of size n;
foreach $i \in [1, n]$ **do**
  $\phi[i] \leftarrow f_i(t)$;
end
return ϕ

To improve the granularity of the fitness function, an approach that has multiple objectives can be used, called *Many-Objective Sorting Algorithm (MOSA)* [22]. In this case, *many objectives* refers to the multiple fitness values begin calculated per individual, one fitness value per goal. Note

that an individual in the MOSA approach is not a suite of programs but rather a single program. As shown in Algorithm 2, instead of reducing all single fitnesses to a single value, MOSA calculates a *fitness vector* of size n, with one fitness per goal to cover. Given the set of all single programs $\mathcal{I}_s$, this yields the fitness function

$$f_{mosa} : \mathcal{F}^n \times \mathcal{I}_s \to [0,1]^n \tag{2.3.3}$$

Having a fitness vector entails that comparing the fitness of two solution candidates cannot be done by just comparing two numbers as with single-objective algorithms. Rather, the notations of *Pareto dominance* and *Pareto optimality* [6] have to be used to compare fitness vectors. Informally, a vector x *dominates* another vector y, if and only if x has a better value for at least one element and not worse values for the remaining elements. A vector z is *optimal*, if and only if there is no other vector that dominates z.

Consequently, a multi-objective algorithm can lead to many different optimal solutions. When covering multiple goals, we therefore want to reduce all optimal solutions into a single suite that covers all goals that are also covered by the individual solutions.

There are more possible fitness functions used by different GAs, for example the *Many Independent Objective (MIO)* approach [2] that maintains one population per goal instead of one population overall. However, these are not further discussed, because we did not consider them in detail for WhiteSyn.

3 Design

WhiteSyn is designed to achieve goal ***G1*** and goal ***G2*** described in Section 1.1. For this purpose, WhiteSyn has two modes, *LoopFinding* mode for ***G1*** and *RedundancyFinding* mode for ***G2***. These modes make use of various components that can be chained together in a modular fashion, as described in the following sections.

3.1 Overview

WhiteSyn consists of the following components:

- **Initialization**: Setup, input verification, static analysis, build CFGs, find loops
- **Instrumentation**: Instrument classes for profiling
- **Input generation**: Generate input programs
  - **Path finding**: Find paths of interest
  - **Input finding**: Given paths, generate input that triggers them
- **Redundancy finding**: Given input, check if it triggers redundancies

Figure 3.1 and Figure 3.2 show, how the components are used in *LoopFinding* mode and *RedundancyFinding* mode, respectively.

3.2 Initialization

The initialization component is the first component that has to be executed to properly use WhiteSyn. It validates the input, sets up the bytecode instrumentation framework, builds the basic blocks and the resulting CFGs (see Section 2.1) and finds loops. Additionally, it enriches the

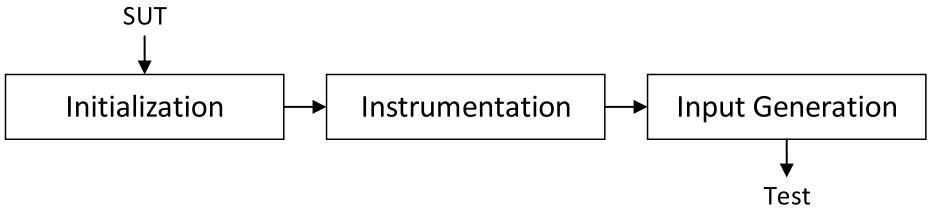


Figure 3.1: The flow through WhiteSyn in *LoopFinding* mode.

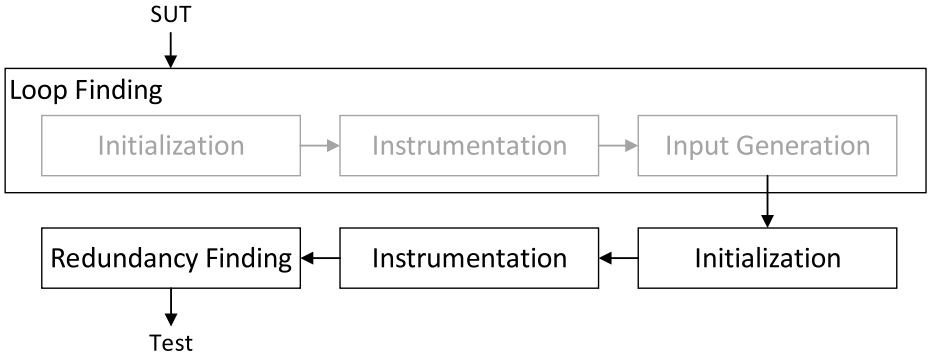


Figure 3.2: The flow through WhiteSyn in *RedundancyFinding* mode.

basic blocks with properties retrieved from a static analysis that can be used later, e.g. if they are part of a loop or how deep in a loop they are. To indicate control flow between methods, designated blocks are inserted at the corresponding call sites in the CFGs. Note that we cannot connect CFGs of different methods to form one global CFG, because this would result in an enormous amount of required memory, since each method call site would need its own copy of the called CFG[1].

3.3 Instrumentation

The instrumentation component adds profiling instructions to the provided bytecode, for which the bytecode instrumentation framework Soot [29] is used. Depending on the mode WhiteSyn is executed in, different profiling instructions are required. For both modes, a statement that emits the corresponding block identifier is added at the start of each basic block. This is critical for any application of WhiteSyn, since it allows tracking the flow of an execution through the SUT. *RedundancyFinding* mode requires additional profiling instructions to be added to track loop iterations, as further described in Section 3.6. A dedicated *profiler* class collects all profiling information and emits them via the standard output or a file.

After instrumenting and validating the bytecode, the instrumentation component writes the modified classes, along with dependent unmodified classes, to .class files, using Soot's ASM backend [4]. When instrumenting JDK code, it is additionally required to repackage all java.* classes, since otherwise, problems with Java reflection arise during input finding. For this purpose, a tool named *JarJar*[2] is used that renames all java.* classes and the corresponding references in other classes.

3.4 Input Generation

This component is the core of WhiteSyn. Given a SUT, it generates the input that triggers the required control-flow through the SUT. Thus, first, some candidate paths that correspond to desired behavior need to be found. Afterwards, an input program that triggers these desired paths

[1] If there was only one CFG per method in a global CFG, all call sites to this method from anywhere in the system would be connected through the called method's CFG. This is not wrong (a CFG may contain infeasible transitions) but highly undesirable, since it leads to many paths that contain infeasible transitions and are therefore infeasible as a whole.

[2] `https://github.com/pantsbuild/jarjar`

needs to be generated. Both of these problems are non-trivial and therefore explained separately in the following subsections.

3.4.1 Path Finding

Starting from a known block (the entry block of the method under test) and given the minimum loop depth δ, WhiteSyn searches for paths that reach loop depth δ. If the SUT is larger than a single method, the entry blocks of all valid[3] methods from the SUT are considered as possible start blocks. A search is started from each such start block, one after the other.

If we only consider loop depth on a per-method level, finding paths that reach loop depth δ is straight-forward. Iterate through all basic blocks and check their loop depth within the method: if it is $\geq \delta$, find a path from the start block to this end block and the task is completed.

However, for many nested loops, their nesting spans over more than one method due to method calls. Consider the motivating example in Listing 1.1. A loop depth of 2 is only achieved, if a `contains` method that executes a loop is called on `Line 8`, e.g. if `col2` is of dynamic type `ArrayList`. Nesting across methods may also be more prone to performance bugs, since the nesting might not be apparent in some cases, or not even present at the time of coding, if we consider virtual dispatch (one overriding method might have a loop, while another does not, as is shown in Listing 1.1). To address this issue, all paths that achieve the minimum loop depth δ from a certain start block to all candidate end blocks have to be considered. A candidate end block is a backjump[4] in a loop. It is therefore guaranteed that at least one iteration in this loop is triggered by a path that reaches the backjump. We observed that the number of such paths might be in the millions.

We designed an algorithm that retrieves all desired paths, as described before, that do not contain any block more than once, i.e. a loop-free and recursion-free path. This facilitates the path finding. However, we later need to accommodate the minimum number of loop iterations μ separately in the fitness function, whereas the minimum loop depth δ is already encoded in the paths found.

Algorithm 3 shows how these paths are retrieved. It applies a depth-first search, starting from a start block S. A stack is used to keep track of the current state of the search. An element in the stack contains the block b, its index i in the path and the loop depth d. Throughout the search, variable *path* contains the current progress of the path finding up to index i after adding b to *path*. Note that *path* might contain blocks from previous steps of the search after the index i. This is intended because reusing a single array of blocks for keeping track of the current path saves on memory. We additionally need a map to efficiently check which blocks are currently in the path, to prevent any duplicates from being added. As mentioned before, we want a loop-free path to the end block candidate, since achieving the desired number of loop iterations is handled separately, as explained in Section 3.5.

When a new block b has successfully been added to *path*, all of its successors are added to the stack, considering the depth differences between them. Additionally, if b calls another method, the start block of the called method is also added to the stack *without* adjusting the loop depth d. This

[3]Public, in a public class, not abstract, not native, not a constructor nor static initializer
[4]The last block in a loop body before jumping back to the loop head

Algorithm 3: The algorithm to retrieve all paths P that reach a least loop depth δ from a start block S to a set of candidate end blocks E.

Name: getPaths
Input: Start Block S, set of candidate end blocks E, minimal loop depth δ
Output: Set of paths P, starting from S, ending in a block of E and reaching at least loop depth δ

```
P ← empty list;
path ← empty path;
map ← empty map (block → index); // Maps a block to its index in path
todo ← empty stack; // Contains tuples (block, index, depth)
todo.push(S, 0, 0);
while not todo.isEmpty() do
    top ← todo.pop();
    b ← top.block(); // Block to potentially add to path
    i ← top.index(); // Index of b in path, if b is added later
    d ← top.depth(); // Current loop depth
    if map.contains(b) and map.get(b) < i then
        // Skip entry, if b is already in path
        continue;
    end
    path.add(b);
    map.put(b, i);
    if E.contains(b) and d ≥ δ then
        // Add the newly found valid path to the result
        p_new ← path.cloneSubpath(0, i); // 0 and i are inclusive
        P.add(p_new);
    end
    if b.callsMethod() then
        // Follow the method call
        b_entry ← b.calledMethod().entryBlock();
        todo.push(b_entry, i + 1, d); // Preserve d over method call
    end
    // Follow all successors of b
    foreach s in b.successors() do
        Δ ← s.depth() − b.depth(); // Depth difference
        d_new ← d + Δ; // Depth after following s
        todo.push(s, i + 1, d_new);
    end
end
return P;
```

is important, such that the loop depth is preserved over method calls, e.g. when calling a method inside of a loop.

When running Algorithm 3, a block might be visited multiple times, which is intended. In fact, a block is visited once for every possible path from S to the block. Naturally, this might lead to a huge amount of paths recorded, especially when dealing with interfaces with many implementations or a lot of conditionals. However, out of the possibly millions of different paths from a start block S to a candidate end block, most of these paths do not differ by blocks that help guiding the genetic algorithm towards a test that reaches the path's end block during input finding. To reduce the number of paths that need to be covered, we aim to reduce all paths between the same two start block and end block to a single path, containing only blocks that *dominate*[5] all blocks in all paths between these two blocks. We will refer to the reduced set of paths as a set of *optimal paths*, i.e. paths that WhiteSyn aims to *cover* (find input, s.t. the control flow follows the path) during input finding.

To illustrate this procedure, consider Figure 2.1a. If we target block 5, there are two loop-free paths to reach it: $\{0, 1, 2, 3, 5\}$ and $\{0, 1, 2, 3, 4, 5\}$. However, the branch block 4 of the conditional does not contribute anything to the path that makes it easier to find target block 5. Rather, if the path we aim to cover contains branch block 4, it potentially makes finding a path to block 5 more difficult, because if block 5 was reached but without taking the branch, the path would not be considered covered. If an optimal path even contained an infeasible branch, it would not be possible to cover this path, even though the target block might be reached. To prevent this, we use the path of all dominating blocks over all paths from the start to the end block as the optimal path. In this case this would be $\{0, 1, 2, 3, 5\}$.

It is important to note that in general, not all paths that are found during path finding are feasible. Since a CFG might contain infeasible edges, a path might include such an edge, making the entire path infeasible.

3.4.2 Input Finding

Having found all optimal paths that should be covered, the input to trigger them is generated. For this purpose, EvoSuite [8] is used. EvoSuite is a test generation tool that generates JUnit tests to achieve high branch coverage, using a genetic algorithm (see Section 2.2). However, EvoSuite can be adjusted for our purpose of generating input that covers predefined paths, by providing corresponding fitness functions.

The individuals for the genetic algorithm are input programs to the SUT, represented by a list of statements, and the population is a set of such programs. A crossover operation exchanges parts of two programs at a random point in the statements list. A mutation can be applied to a program by altering its statements, adding new ones or removing or multiplying existing ones. Most of these techniques are rather generic and can be used out of the box from EvoSuite. The main component of the genetic algorithm of EvoSuite that has to be adapted to our needs is the fitness function.

Since we want to generate input that triggers a predefined path, it is natural to make the fitness function a measurement of how many blocks of the optimal path (the path we want to cover) are in

[5] A block b_1 dominates another block b_2, if all paths from the start block S to b_2 go through b_1

the actual path (the path that the program in question triggers). The more blocks of the optimal path are covered by the actual path, the lower (better) the fitness and if all blocks of the optimal path are covered, the fitness is 0. This approach works fine, if μ (the minimum number of loop iterations of all involved loops) is 1. However, if $\mu > 1$, we have to make sure that the targeted loop is actually executed as many times as required. If additionally we aim to cover nested loops, i.e. $\delta > 1$, an inner loop must be executed at least μ times in *every* iteration of its outer loop (as described in Section 3.5).

We define custom fitness functions, such that the genetic algorithm is able to find input that triggers both the optimal paths previously found and the number of required loop iterations. The lower the fitness of a generated input, the closer the generated input program is to trigger the desired path and iterate the required loops. To evaluate a test program, the profiler output of its execution, i.e. the actual path that the execution triggered, is used, as illustrated in Section 3.5.

3.5 Fitness Functions

The fitness of a generated input drives the evolution and is therefore vital to the generation of useful programs. We need to differentiate between the fitness function f_s implemented in WhiteSyn that calculates the fitness for a single goal (defined by the optimal path and the minimum number of loop iterations), and the fitness functions f_{ws} or f_{mosa}, respectively, that combine single fitness functions to get a test suite that covers multiple goals.

Let Ω be the the set of all possible paths. Inspired by Function 2.3.1, we define the single fitness function as

$$\begin{aligned} &f_s : \Omega \to [0, 1] \\ &f_s(\alpha) := \phi \\ &\text{where } \alpha \in \Omega \end{aligned} \tag{3.5.1}$$

f_s is defined per optimal path θ and the minimal loop iterations parameter μ and takes an actual path $\alpha \in \Omega$ as argument. The minimal loop depth parameter δ is already encoded in the optimal path and therefore does not have to be considered separately.

The calculation of the fitness is split into two parts: On the one hand, the *path fitness* ρ indicates how close α is to θ, regardless of any loop iterations. As explained in Section 3.4.1, an optimal does not contain any block more than once, even if $\mu > 1$. On the other hand, to achieve the minimal number of loop iterations of all loops in question, we also need the *iteration fitness* η, which represents how many iterations of the required loop(s) α triggers. To combine ρ and η, an additional weighing parameter $\omega = 0.25$ is introduced, which represents the weight of the path fitness ρ. The next paragraphs explain the calculation of the two fitnesses ρ and η and the overall fitness ϕ. Note that all fitnesses are normalized.

Algorithm 4 illustrates how to calculate the path fitness ρ. It represents the number of blocks in θ that are not in α as a fraction of the total size of θ.

To calculate the iteration fitness, let us first define the terms *loop execution* and *loop iteration*.

Algorithm 4: The algorithm to calculate the path fitness ρ for an actual path α.

Name: getPathFitness
Input: Actual path α, optimal path θ
Output: Path fitness $\rho \in [0, 1]$
$m \leftarrow \theta.size()$;
foreach *Block* b *in* θ **do**
  if $b \in \alpha$ **then**
    $m \leftarrow m - 1$;
  end
end
$\rho \leftarrow m/\theta.size()$; `// Normalize`
return ρ

A loop execution is a distinct execution of a loop statement, e.g. a `while` statement. Each loop execution can have zero or more loop iterations. In turn, a program might execute the same loop zero or more times, e.g. by calling a method containing a loop multiple times.

Algorithm 5: The algorithm to calculate the number of iterations of a single loop. Method call `ex.noIterations()` retrieves the number of loop iterations of loop execution `ex`.

Name: getIterationsSingleLoop
Input: Actual path α, loop λ
Output: Most iteratitions n of any execution of loop λ
$n \leftarrow 0$;
`// Iterate through each execution of the loop`
foreach ex *in* $\lambda.executions(\alpha)$ **do**
  $n \leftarrow max(n, ex.noIterations())$;
end
return n;

To calculate the iteration fitness, we first have to determine, how many loops are involved. For simplicity reasons, we restrict this number to be 1 or 2. For both cases, we need to find out how many times the loop(s) iterates(s). If only one loop is involved, this is trivially done by analyzing all loop executions and recording the maximum number of loop iterations of any loop execution, as shown in Algorithm 5.

However, handling two nested loops (a *double loop*) is more complicated. A tuple (n_{outer}, n_{inner}), called *iteration tuple*, must guarantee that for at least one outer loop execution, the outer loop iterates n_{outer} times and the inner loop iterates n_{inner} times in *every* iteration of the outer loop. This is important, because if the inner loop did not iterate n_{inner} times in every outer loop iteration, a program that triggers the outer loop to iterate μ times and the inner loop to iterate μ times in *only one* iteration of the outer loop, would be considered optimal in terms of iteration fitness. This is not the case because, the iterations of the inner loop are not amplified by the nesting, if the inner loop is only fully iterated once.

Consider a use case of the motivating example in Listing 3.1. There are three different lists,

Listing 3.1: A use case of the motivating example from Listing 1.1.

```
1  void test() {
2     List list1 = new ArrayList();
3     List list2 = new ArrayList();
4     List list3 = new ArrayList();
5
6     //Fill list1 with 3 different elements
7     list1.add("A"); list1.add("B"); list1.add("C");
8
9     //Fill list2 with 2 elements from list1 and one different element
10    list2.add("A"); list2.add("B"); list2.add("D");
11
12    //Fill list3, s.t. it is disjoint from list1
13    list3.add("X"); list3.add("Y"); list3.add("Z");
14
15    //The inner loop is only iterated three times in one iteration of the outer loop
16    Utils.subtract(list1, list2);
17
18    //The inner loop is iterated three times in all three iterations of the outer loop
19    Utils.subtract(list1, list3);
20 }
```

`list1`, `list2` and `list3`, each of dynamic type `ArrayList`. Therefore, their implementation of `contains` has a loop (as shown in Listing 1.3) and calling `subtract` triggers a double loop, if both arguments of `subtract` are non-empty. The number of iterations of the outer loop (in `subtract`) is always the same as the number of elements in the first argument, regardless of the actual contents of the lists. On the other hand, the inner loop (in `contains`) only iterates until the element is found or the end of the list is reached. Therefore, to trigger an execution that iterates the inner loop the maximal number of times (three in our example) in every outer loop iteration, the two lists provided as arguments to `subtract` must be disjoint.

In Listing 3.1, `list1` contains three different elements `"A"`, `"B"` and `"C"`, `list2` also contains elements `"A"` and `"B"`, as well as a different element `"D"` and `list3` contains three elements that are neither in `list1` nor in `list2`. Consider `Line 16` in Listing 3.1. Since the first two elements (`"A"` and `"B"`) of `list2` are also in `list1`, the inner loop does not iterate the full three times in the first two iterations of the outer loop. Only when the outer loop checks for element `"C"`, the inner loop does the full three iterations, because `list2` does not contain `"C"`. The call on `Line 16` therefore leads to an iteration tuple $(n_{outer}, n_{inner}) = (3, 0)$, since `contains` returns before completing the first iteration of the inner loop when `"A"` is provided as argument. On the other hand, on `Line 19`, the inner loop iterates three times, every time `contains` is called, because `list1` and `list3` are disjoint. Therefore, the iteration tuple is $(n_{outer}, n_{inner}) = (3, 3)$, i.e. the inner loop is guaranteed to be iterated three times in each outer loop iteration.

Algorithm 6 shows how to retrieve the best iteration tuple. We consider inner iterations to be more important than outer iterations, therefore iteration tuples are first compared by their inner iterations (more is better) and only if they are equal by their outer iterations (also more is better).

Let us break down Algorithm 6, starting with its outermost loop. This loop iterates over all executions of λ_{outer} to find the best iteration tuple. If there are other executions of λ_{outer} that

Algorithm 6: The algorithm to determine the best (most inner iterations, and on equality, most outer iterations) iteration tuple (n_{outer}, n_{inner}). This algorithm guarantees that in at least one outer loop execution, the outer loop is iterated n_{outer} times and the inner loop is iterated n_{inner} times in *every* iteration of the outer loop. Method call `ex.noIterations()` retrieves the number of loop iterations of loop execution `ex`.

Name: getIterationsTwoLoops
Input: Actual path α, outer loop λ_{outer}, inner loop λ_{inner}
Output: Best iteration tuple $t = (n_{outer}, n_{inner})$

```
t ← (0, 0) // The initial tuple
// Iterate through each execution of the outer loop in path α
foreach ex_outer in λ_outer.executions(α) do
    n_inner ← ∞;
    // Iterate through each iteration of the outer loop's execution
    foreach it_outer in ex_outer.iterations() do
        executions_inner ← inner executions within it_outer;
        maxIt_inner ← 0;
        // Find the inner loop execution with the most iterations
        foreach ex_inner in executions_inner do
            iterations_inner ← ex_inner.noIterations();
            maxIt_inner ← max(maxIt_inner, iterations_inner);
        end
        n_inner ← min(n_inner, maxIt_inner);
    end
    if n_inner == ∞ then
        n_inner ← 0;
    end
    n_outer ← exec_outer.noIterations();
    t_new ← (n_outer, n_inner);
    if t_new is better than t then
        t ← t_new;
    end
end
return t;
```

produce inferior tuples, they are ignored. The middle loop may seem counterintuitive at first. It iterates over the iterations of the outer execution and searches the *minimum number of guaranteed iterations of* λ_{inner}. We need the minimum, such that this guarantee holds, otherwise we run into the problem described above with Listing 3.1. The innermost loop iterates through all executions of λ_{inner} within the current outer loop iteration to find the best one, i.e. the one that iterates the most. We can take the max here, since within the iteration of λ_{outer}, the iterations of every inner execution are guaranteed to occur.

Algorithm 7: The algorithm to calculate the iteration fitness η for a recorded path α.

Name: getIterationFitness
Input: Actual path α, optimal path θ, minimal number of loop iterations μ, involved loops Λ
Output: Iteration fitness $\eta \in [0, 1]$
if $\Lambda.size() == 1$ **then**
  $n \leftarrow getIterationsSingleLoop(\alpha, \Lambda.getSingleLoop())$;
  `// Iterations larger than` μ `distort normalization`
  $missing \leftarrow min(n, \mu)$;
  $\eta \leftarrow missing/\mu$; `// Normalize`
end
else if $\Lambda.size() == 2$ **then**
  $(n_{outer}, n_{inner}) \leftarrow getIterationsTwoLoops(\alpha, \Lambda.outer(), \Lambda.inner())$;
  `// Iterations larger than` μ `distort normalization`
  $n_{outer} \leftarrow min(n_{outer}, \mu)$;
  $n_{inner} \leftarrow min(n_{inner}, \mu)$;
  `// Weigh inner iterations more`
  $missing \leftarrow (\mu + \mu^2) - (n_{outer} + \mu * n_{inner})$;
  $\eta \leftarrow missing/(\mu + \mu^2)$; `// Normalize`
end
else
  `// Unsupported`
end
return η;

Having found the best iteration tuple of an actual path α, we can finally calculate the iteration fitness, as shown in Algorithm 7. If only a single loop is considered, the fitness is represented as a fraction of missing iterations and the number of required iterations μ. If two loops are considered, we weigh the inner iterations quadratically more. The fitness calculation is similar as with a single loop, i.e. a fraction of missing iterations and the number of required iterations, just with adjusted number of missing and required iterations for the inner loop.

Currently, WhiteSyn does not support nesting level three or higher, if $\mu > 1$. However, implementing it would just be a generalization of the approach described above. Note that this does not mean that a triple nested loop is not considered at all, if $\delta = 2$ and $\mu > 1$; it is then just considered as two double loops.

As illustrated in Algorithm 8, once the path fitness and iteration fitness are calculated, the final step is to weigh them to control their importance. Since both fitnesses are normalized, this

Algorithm 8: How to calculate the fitness ϕ for a recorded path α.

Name: getFitness
Input: Actual path α, optimal path θ, minimal number of loop iterations μ, path fitness weight ω, involved loops Λ
Output: Fitness $\phi \in [0, 1]$

```
ρ ← getPathFitness(α, θ);
if μ == 1 then
    // No iteration fitness required
    return ρ;
end
η ← getIterationFitness(α, θ, μ, Λ);
φ ← ω * ρ + (1 − ω) * η; // Weigh the two fitnesses
return φ;
```

is trivial. Note that if $\mu = 1$, no iteration fitness is required, since the optimal path θ already guarantees at least one iteration of the targeted loop.

If we were only interested in finding a single input program for a single goal, the fitness ϕ derived so far would suffice. However, as explained in Section 2.3, we usually want to cover many goals simultaneously. Therefore, we need a way to combine single fitness functions into a fitness function for multiple goals and we must be able to compare two solution candidates and determine, if one is better than the other, to drive the evolution. As in Section 2.3, we consider the two approaches outlined in Function 2.3.2 and Function 2.3.3.

Given the single fitness Function 3.5.1, the number of goals n and the size of a test suite k, we define the fitness function for the whole suite approach as

$$
\begin{aligned}
&f_{ws} : \mathcal{F}^n \times \Omega^k \to [0, 1] \\
&f_{ws}(F, A) := \frac{1}{n} \sum_{i=1}^{n} \min_j f_s^i(\alpha_j) \\
&\text{where } F = [f_s^1, ..., f_s^n] \in \mathcal{F}^n, A = [\alpha_1, ..., \alpha_k] \in \Omega^k
\end{aligned}
\tag{3.5.2}
$$

The major disadvantage of the WS approach is the difficulty to get an expressive fitness for an entire test suite, i.e. reflecting the differences of the fitnesses of the individual goals. The MOSA approach addresses this issue by keeping the single fitnesses for each goal to cover as a fitness vector. Its fitness function is defined as

$$
\begin{aligned}
&f_{mosa} : \mathcal{F}^n \times \Omega \to [0, 1]^n \\
&f_{mosa}(F, \alpha) := [f_s^1(\alpha), ..., f_s^n(\alpha)] \\
&\text{where } F = [f_s^1, ..., f_s^n] \in \mathcal{F}^n, \alpha \in \Omega
\end{aligned}
\tag{3.5.3}
$$

This allows the comparison between individuals to include more information, i.e. all single fitnesses. As mentioned before, the individuals in the WS approach are test suites, whereas the

individuals in MOSA are single tests. Therefore the fitness function of WS takes k paths as input (one for each program in the test suite), whereas the fitness function of MOSA only takes a single path as input. Both approaches are implemented in EvoSuite. Testing has shown that MOSA performs significantly better than WS. The MIO approach (see Section 2.3) that extends the idea of many independent goals, by e.g. keeping a designated population for each goal, is also implemented in EvoSuite. However, it did not perform as well as MOSA during testing.

3.6 Redundancy Finding

Finding redundancy presumes an input to the SUT. Given this input, one can instrument the SUT accordingly, execute the input and analyze the output of the profiler. We use the Toddler oracle that is described in detail in [18], so we will only give a brief overview here.

Toddler is an oracle to find performance bugs. It focuses on performance bugs in nested loops, because they usually have a more severe performance impact than performance bugs in non-nested loops, due to the amplification caused by the nesting. The bugs Toddler is able to catch cause repeated memory-access patterns throughout loop iterations. These repeating patterns might indicate that some memory-accesses are redundant and could e.g. be hoisted out of the loop. To detect such patterns, the profiler needs to track every memory read by storing the value read and the current stack trace. Additionally, profiling instructions have to be added such that each loop start, loop end and loop iteration is recorded.

Given this data, one can create an oracle by analyzing the patterns read within a loop iteration and detect repeating patterns. Detecting such patterns is achieved by comparing the sequences of tuples (value read, stack trace) for each iteration of the loop. If repeating patterns are found, it might be an indication of redundancy.

Toddler does not provide unit test generation, but rather "only" an oracle. It relies on input to the SUT to determine, whether there are redundancies. As illustrated in Figure 3.2, to automatically find that input, we can re-use a test generated by *LoopFinding* mode, by designing the fitness function of the genetic algorithm accordingly.

We restrict ourselves to find redundancies in double loops, since, as discussed, these are more impactful than redundancies in single loops. Therefore, the initial input for Toddler must trigger such a double loop and iterate both loops a minimum number of times (Toddler sets this number to 7). To generate the initial input, we run WhiteSyn in *LoopFinding* mode with the parameters $\mu \geq 7$ and $\delta = 2$.

Given the generated input that, by design of the fitness function, iterates double loops μ times, one can instrument the SUT with the outlined Toddler instrumentation, execute the instrumented input and analyze the output of the profiler. If redundancies are found, we therefore have the input program that triggers them. For proper reporting, the input program is enriched with corresponding comments for the user to refer to.

3.7 Restricting JDK Interface Implementations

During testing, we observed that when applying WhiteSyn to methods that use JDK interfaces as arguments, the runtime noticeably increased. We therefore restricted the allowed interface implementations and disallowed some methods from the JDK.

Restricting JDK interfaces means that only a whitelist of implementations of certain JDK interfaces are allowed to be used in path finding (e.g. `LinkedList` when `Collection` is declared). This *drastically* reduces the number of paths found in path finding and subsequently the number of goals that input finding tries to cover. The issues that arise when having too many paths, and therefore goals, are further discussed in Section 4.4.

Forbidding some JDK methods has similar motivations. Many methods from e.g. `Collection` do not have an impact on iterating any loops, but since there are many implementations of `Collection`, calls to implementations of these methods (e.g. `size`) are frequently generated by the genetic algorithm, wasting time without improving the fitness. Consequently, such forbidden methods do not need to be considered in path finding either.

Testing has shown that in almost all cases, these restrictions do not impact WhiteSyn's ability to generate long running programs or find redundancies. However, some tests suffer from *over-restriction*, i.e. the restriction of JDK interfaces hinders the performance of WhiteSyn, as is shown in Section 4.

4 Evaluation

We evaluate WhiteSyn in regard to the two goals ***G1*** and ***G2*** defined in Section 1.1. Since input finding is the longest running component of WhiteSyn, throughout this section, we refer to timeouts in regard of the timeout of the input finding component. While path finding might also take a considerable amount of time, it is usually much lower than the time spent finding input. All other components, e.g. instrumenting or finding redundancies, have a negligible runtime compared to path finding and input finding. Since WhiteSyn is not deterministic, all experiments in Section 4.1 and Section 4.2 were repeated 5 times.

For the evaluation, we used methods from the following libraries: Apache Commons Collections (ACC), Apache Commons Lang (ACL), Apache Commons Math (ACM), Apache Ant, Google Guava and the JDK.

All experiments were run on a machine with Ubuntu 16.04.1 LTS installed, having two Intel Xeon E5-2699v4 CPUs (22 cores each, hyper-threading disabled) with 504 GB of RAM. Each experiment ran in a Docker[1] image with WhiteSyn having 16 GB of memory and EvoSuite having 20 GB of memory, respectively. WhiteSyn and EvoSuite usually require much less memory, but to eliminate running out of memory as a variable, it was set rather high. Up to 12 Docker images were run in parallel.

4.1 Loop Finding

As desired with ***G1***, we aim to find long running programs, to which we refer to in terms of the number of loop iterations triggered. For the evaluation, we consider the bottlenecks found by PerfSyn [27] and the two methods `contains` and `containsAll` from JDK's `List`. In total, we evaluated *LoopFinding* mode with 20 tests.

Consider Table 4.1 for an overview of the bottlenecks found by PerfSyn. Each test corresponds to a method under test and a version of the library it is contained in. Additionally, Table 4.1 indicates, if (and why) a bottleneck is not findable by WhiteSyn or if it suffers from over-restriction (see Section 3.7). For technical reasons, we did not consider the bottlenecks with ID 21 and ID 22 discovered by PerfSyn. Throughout this chapter, we will refer to the tests by their ID from Table 4.1. We aim to generate tests that trigger many iterations of single loops, i.e. $\delta = 1$. Therefore, only bottlenecks that might reach loops are considered in the evaluation. Since more loop iterations correspond to longer running programs, we varied the loop iteration parameter μ from 16 to 1024.

[1] `https://www.docker.com/`

ID	Library	Version	Method	Not findable	Over-restricted	Findable	Comment
perfsyn_1	ACL	3.5	ArrayUtils.removeAll			✓	
perfsyn_2	ACL	3.4	ArrayUtils.removeElements			✓	
perfsyn_3	ACL	3.4	ArrayUtils.indexOf			✓	
perfsyn_4	ACL	3.5	ArrayUtils.isNotEmpty	✓			No loop
perfsyn_5	ACL	3.4	ArrayUtils.isSameLength	✓			No loop
perfsyn_6	ACL	3.4	CharSetUtils.squeeze			✓	
perfsyn_7	ACL	3.4	StringUtils.getLevenDist			✓	
perfsyn_8	Ant	1.9.4	IdentityStack.containsAll			✓	
perfsyn_9	Ant	1.9.4	IdentityStack.removeAll			✓	
perfsyn_10	Ant	1.9.4	IdentityStack.retainAll			✓	
perfsyn_11	ACC	4.1	ListOrderedMap.remove			✓	
perfsyn_12	ACC	3.2.1	ListOrderedSet.addAll			✓	
perfsyn_13	ACC	4.1	ListOrderedSet.addAllAtIndex			✓	
perfsyn_14	ACC	3.2.1	ListOrderedSet.toArray			✓	
perfsyn_15	ACC	3.2.1	ListOrderedSet.remove			✓	
perfsyn_16	Ant	1.9.4	VectorSet.addAll			✓	
perfsyn_17	Ant	1.9.1	VectorSet.clone		✓		
perfsyn_18	ACC	3.2.1	TreeList.addAll		✓		
perfsyn_19	ACC	3.2.1	TreeList.addAllAtIndex			✓	
perfsyn_20	ACM	3.6	EnumIntDist.probability			✓	

Table 4.1: The bottlenecks reported by PerfSyn and whether it is possible to find them with WhiteSyn and if they suffer from over-restriction in WhiteSyn. If applicable, the last column indicates why it is not possible to find a bottleneck.

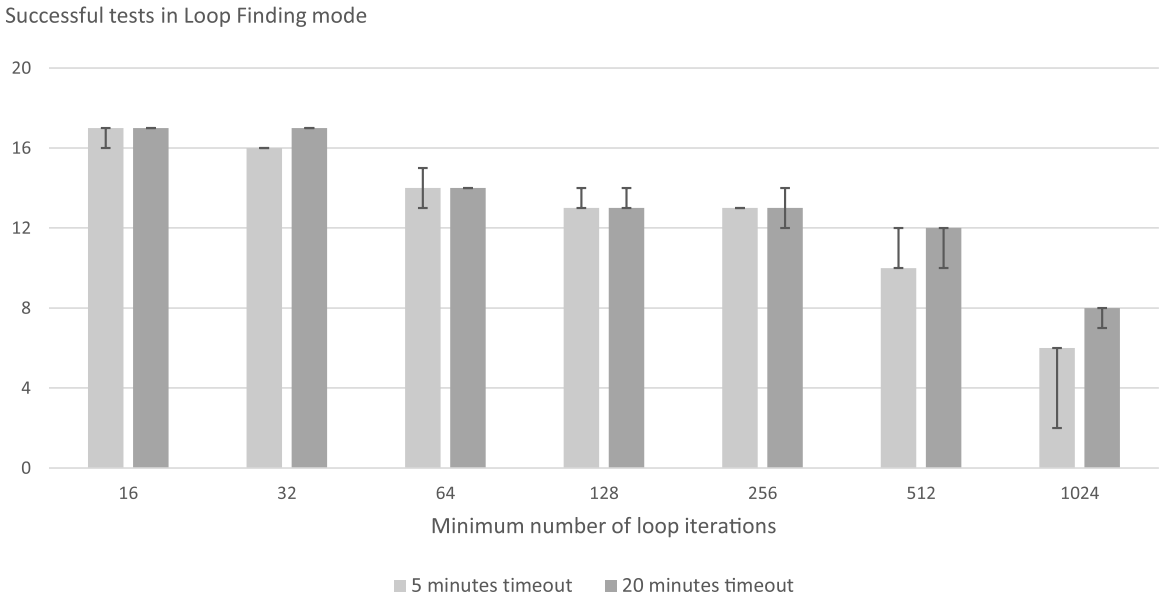


Figure 4.1: The median across 5 runs of the number of successful tests (out of 20 tests) when running WhiteSyn in *LoopFinding* mode for different values of μ. The error bars indicate the minimum and the maximum number of successful tests of a single run. All 18 tests described in Table 4.1 that are marked as 'over-restricted' or 'findable', as well as the two methods `contains` and `containsAll` of JDK's `List` are considered as tests.

The results for the PerfSyn bottlenecks and the two `List` methods are shown in Figure 4.1. PerfSyn uses a timeout of 5 minutes, therefore we set the input finding timeout to 5 minutes as well. Additionally, we evaluated the tests with an input finding timeout of 20 minutes, to investigate if there are any improvements. However, increasing the timeout did not significantly impact the results.

As Figure 4.1 shows, WhiteSyn is able to reliably generate programs that trigger up to 32 loop iterations. With $\mu = 16$, 17 out of 20 tests are successful and with $\mu = 32$, 16 out of 20 tests are successful. When increasing the timeout from 5 to 20 minutes, $\mu = 32$ also yields 17 successful tests. Between $\mu = 64$ and $\mu = 256$, WhiteSyn successfully generates programs for 13 – 14 out of the 20 tests. With larger value of μ, WhiteSyn struggles to generate programs that iterate the targeted loop the required number of times.

Figure 4.1 illustrates that across different runs of the same experiments, the number of successful tests does not vary by more than ± 1. This is an important result that indicates that it is possible to get a consistent number of successful test generations when using WhiteSyn. The only exception to this is the minimum number of successes 2 for $\mu = 1024$ and a timeout of 5 minutes. However, the minimum is inherently prone to outliers. When inspecting the results, we observed that 4 out of 5 runs achieve 6 successes (i.e. the median and the maximum are also 6) and the minimum of 2 can therefore be considered such an outlier.

Some of the failing tests can be explained by an over-restriction of JDK interfaces (see Section 3.7 and Table 4.1) that prevents paths from being found ('perfsyn_17') or input being generated ('perfsyn_18'). In general, the number of possible statements that may be generated strongly influences the performance of the input finding component, which we discuss in Section 4.5.4.

When investigating the tests that succeed only for small values of μ but fail for medium to large values of μ, two groups are observed. The first group (e.g. 'perfsyn_13') requires statements to be generated that rely on integer sampling, for example an array access. Sampling valid integers is inherently difficult with a genetic algorithm, as further discussed in Section 4.5.2. The second group of tests (e.g. 'perfsyn_11' and 'perfsyn_20') targets a loop that is unlikely to be triggered when randomly generating statements, as with a genetic algorithm. This is addressed in Section 4.5.1.

4.2 Redundancy Finding

To evaluate goal ***G2*** and given that we use the Toddler oracle to find redundancies, we evaluate WhiteSyn's ability to automatically expose the performance bugs reported by Toddler [18]. However, there are certain bugs that WhiteSyn cannot find at all. Table 4.2 lists all reported bugs, if it is possible to find them with WhiteSyn or if they suffer from over-restriction (see Section 3.7). For technical reasons, we did not consider the bug with ID 410 in Apache Commons Collections discovered by Toddler. Each test corresponds to a method under test and a version of the library it is contained in. While some bugs do not affect double loops, which we made a requirement, and some bug descriptions are not available anymore, most notable are the bugs in private or non-static inner classes. Since the final output of WhiteSyn is a test that triggers the redundancies found, we

ID	Library	Version	Method	Not findable	Over-restricted	Findable	Comment
406	ACC	3.2.1	ListUtils. subtract			✓	
407	ACC	3.2.1	ListOrderedSet. removeAll			✓	
408	ACC	3.2.1	SetUniqueList. removeAll			✓	
409	ACC	3.2.1	ListOrderedSet. addAll			✓	
412	ACC	3.2.1	CollectionUtils. subtract			✓	
413	ACC	3.2.1	AbstractDualBidiMap. View.removeAll	✓			Non-static inner class
425	ACC	3.2.1	ListOrderedMap. remove	✓			No double loop
426	ACC	3.2.1	ListOrderedSet. retainAll		✓		
427	ACC	3.2.1	SetUniqueList. retainAll			✓	
429	ACC	3.2.1	MultiValueMap. Values.containsAll	✓			Private inner class
433	ACC	3.2.1	TreeList. addAll	✓			No double loop
434	ACC	3.2.1	CursorableLinkedList. containsAll			✓	
53622	Ant	1.8.4	VectorSet. retainAll			✓	
53637	Ant	1.8.4	VectorSet. addAll	✓			Native inner loop
53803	Ant	1.9.1	IdentityStack. retainAll			✓	
53821	Ant	1.9.1	IdentityStack. removeAll			✓	
53822	Ant	1.9.1	IdentityStack. containsAll			✓	
1013	Guava	12.0	LinkedHashMultimap. SetDecorator.removeAll	✓			Private inner class
1144	Guava	12.0	Iterators. removeAll			✓	
7186403	JDK	1.8.0_181	AbstractCollection. retainAll			✓	
1069	Guava	—	—	✓			Private inner class
1085	Guava	—	—	✓			Private inner class
5739	Groovy	—	—	✓			Bug description N/A
7186422	JDK	—	—	✓			Bug description N/A
3548453	JFC	—	—	✓			Affected version N/A
415	JUnit	—	—	✓			Bug description N/A

Table 4.2: The bugs reported by Toddler and whether it is possible to find them with WhiteSyn and if they suffer from over-restriction in WhiteSyn. If applicable, the last column indicates why it is not possible to find a bug.

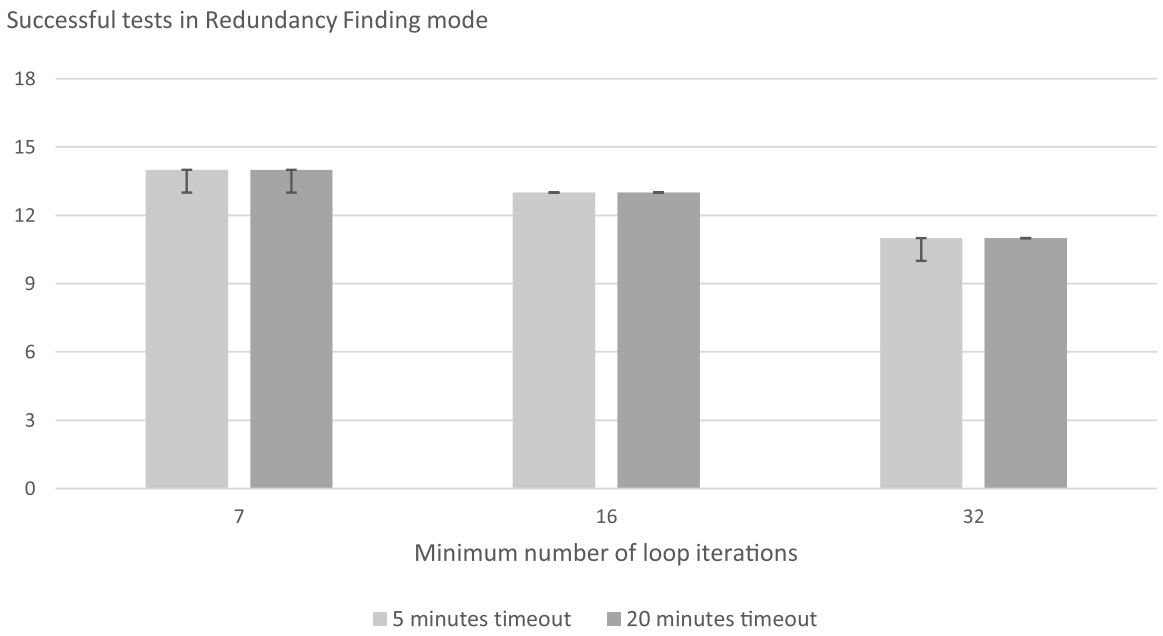


Figure 4.2: The median across 5 runs of the number of successful tests (out of 15 tests) when running WhiteSyn in *RedundancyFinding* mode for different values of μ. The error bars indicate the minimum and the maximum number of successful tests of a single run. All 14 tests described in Table 4.2 that are marked as 'over-restricted' or 'findable', as well as the motivating example from Listing 1.1 are considered as tests.

cannot directly target such inner classes without reflection. These limitations are further discussed in Section 4.5.3. Additional to the Toddler bugs, we added a test for the motivating example from Listing 1.1 and therefore have 15 tests to evaluate. Throughout this chapter, we will refer to the tests by their ID from Table 4.2.

If we consider the Toddler performance bugs for which WhiteSyn might generate tests, Figure 4.2 shows that all but a single one of the redundancies are found for $\mu = 7$. This value is of particular interest, since the redundancy definition of Toddler requires both, the inner and outer, loops to be executed 7 times to even be considered for redundancy analysis. For $\mu = 16$, 13 out of 15 tests are successful and for $\mu = 32$, 11 tests are successful. When comparing the results for an input finding timeout of 5 and 20 minutes, respectively, we note that there are no significant differences. Similar to the results of *LoopFinding* mode, we observe that the number of successful tests does not vary by more than 1 across the different runs, which indicates that WhiteSyn is able to generate tests that trigger redundancies consistently.

For bug '426' of the ACC library, WhiteSyn is unable to find input due to over-restriction of JDK interfaces (see Section 3.7). The remainder of the failures is most likely caused by the same reasons addressed in Section 4.1 and we further discuss these issues in Section 4.5. Note that all failures in shown in Figure 4.2 are caused by the input finding component.

As an illustration, generated input for the motivating example for $\mu = 7$ is shown in Listing 1.5. Listing 4.1 shows the generated input for performance bug '427' of the ACC library from Table 4.2.

Listing 4.1: A generated test for peformance bug '427' of the ACC library with $\mu = 32$ and an input finding timeout of 5 minutes.

```
public void test0() {
   ArrayList<String> arrayList0 = new ArrayList<String>();
   boolean boolean0 = arrayList0.add("(F.-bSQ!C&A!v:6t2");
   //... (30 times the same method call)
   boolean boolean31 = arrayList0.add("(F.-bSQ!C&A!v:6t2");

   ArrayList<Object> arrayList1 = new ArrayList<Object>();
   SetUniqueList setUniqueList0 = SetUniqueList.decorate(arrayList1);

   boolean boolean32 = arrayList1.add((Object) null);
   //... (30 times the same method call)
   boolean boolean63 = arrayList1.add((Object) null);

   //This is the call to the method under test
   boolean boolean64 = setUniqueList0.retainAll(arrayList0);
}
```

4.3 Complex Input

Section 4.1 and Section 4.2 evaluated SUTs that are single methods, mostly with collections as parameters. However, an interesting aspect would be to use WhiteSyn on SUTs that take a more complex input, e.g. an XML file. Unfortunately, preliminary testing has shown that WhiteSyn is not able to generate such input, in particular, if triggering many loop iterations is dependent on a complex object structure and not solely on the arguments of the method under test.

4.4 Large Codebases

WhiteSyn supports generating tests with classes or entire jars (which we refer to as *large codebases*) instead of only single methods as input, but testing has shown that such large input causes various problems. First, finding the initial paths (see Section 3.4.1) can consume a lot of time and memory. To prevent the path finding from taking too long, a timeout for the path finding algorithm was introduced. However, this of course might lead to some paths of interest not being found, because the path finding algorithm may reach its timeout before finding these paths. To prevent memory exhaustion, found paths are reduced to paths of dominating blocks (as explained in Section 3.4.1) regularly *during* the search.

Second, targeting many (dozens or more) goals at once proved to be rather difficult with the MOSA approach. Since it only maintains a single population, the more goals there are, the more difficult it is to actually find individuals that cover them, since the evolution might perform badly due to the many, possibly independent or even contradictory, goals, as discussed in Section 2.3. Also the chances of having infeasible goals are increased with having more goals as well, leading to wasted resources when trying to cover them. The MIO approach might solve these issues, as it is specifically designed to deal with many independent (and possibly infeasible) objectives, but it did not perform better than MOSA in our testing.

Name	Library	Version	Total	Covered MOSA	Covered MIO
commons-collections	ACC	3.2.1	345	6 (0.017)	3 (0.009)
org.apache.ant	Ant	1.9.1	105	2 (0.019)	2 (0.019)
perfSynBottlenecks	—	—	18	1 (0.056)	1 (0.056)
toddlerBugs	—	—	14	12 (0.857)	10 (0.714)

Table 4.3: The number of total goals and covered goals when running WhiteSyn on codebases to find input with the MOSA and the MIO approach, respectively. The numbers in brackets in the covered goals columns are the fraction of the number of covered goals and the total number of goals. Row 'perfSynBottlenecks' and row 'toddlerBugs' refer to the tests from Table 4.1 and Table 4.2, respectively, bundled into a single jar. The remaining tests correspond to a reasonable subset of classes of the respective libraries. The input finding timeout is set to 30 minutes per class and $\mu = 7$.

Listing 4.2: The method under test `ListOrderedMap.remove` of test 'perfsyn_11'.

```
1 public V remove(final Object key) {
2    V result = null;
3    if (decorated().containsKey(key)) {
4       result = decorated().remove(key);
5       insertOrder.remove(key);
6    }
7    return result;
8 }
```

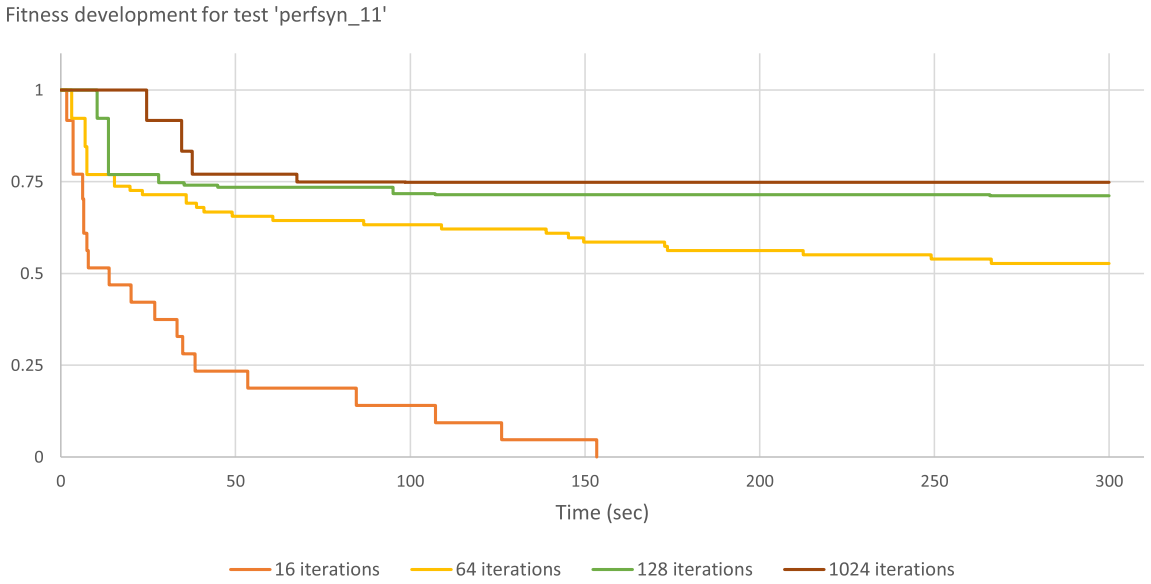
Table 4.3 shows results of preliminary tests to compare the MOSA and MIO approach for large codebases. The rows named 'perfSynBottlenecks' and 'toddlerBugs' are used as a reference and correspond to the tests from Table 4.1 and Table 4.2, respectively, bundled into a single jar. Therefore, running WhiteSyn with these two inputs should yield the same results, as using the individual methods from Table 4.1 and Table 4.2 as input. This is achieved only for the MOSA approach for test 'toddlerBugs', whereas for test 'perfSynBottlenecks', almost no input is found.

When using a reasonable subset of classes of the libraries ACC 3.2.1 and Ant 1.9.1 (which many of the tests from Sections 4.1 and 4.2 are contained in) as input to WhiteSyn , we expect at least the same bottlenecks and bugs to be found, as when using the individual methods as input. However, again, almost no input is found. This is most likely caused by targeting too many goals at once (i.e. trying to cover too many paths at once), which leads to the genetic algorithm not generating any tests at all.

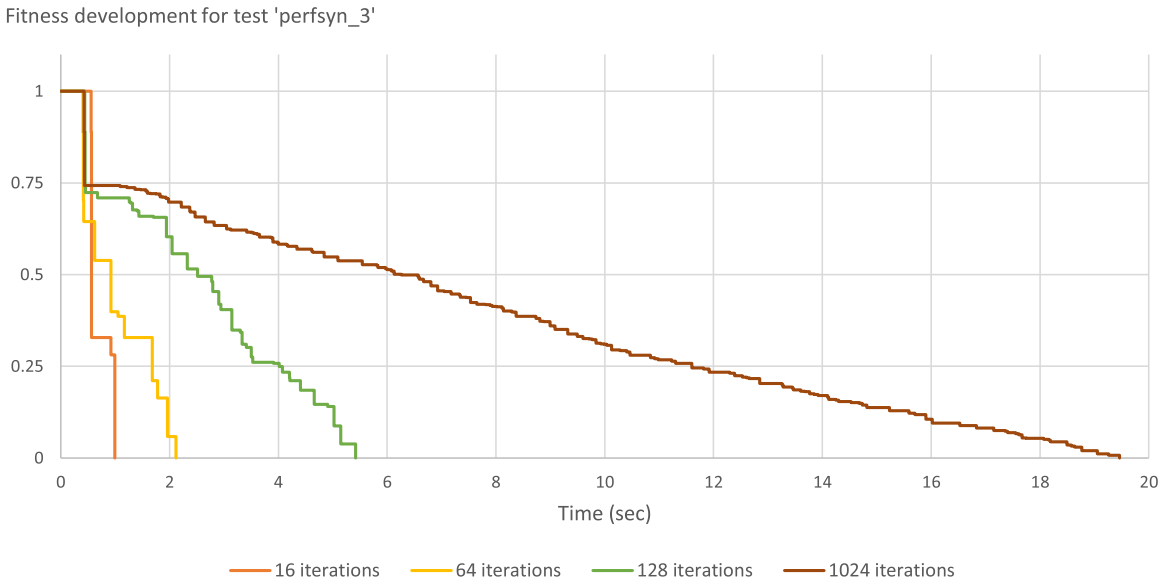
4.5 Discussion

4.5.1 Limitations of Genetic Algorithms

As mentioned in Section 4.1, some tests target loops that are unlikely to be covered by input generated by a genetic algorithm. Consider Figure 4.3a that illustrates the development of the fitness during input finding for different values of μ for test 'perfsyn_11'. For $\mu = 16$, an input that covers the desired goal is found. However, for larger values of μ, no input is generated. After an initial improvement of the fitness to approximately 0.75, the fitness is not significantly improved



(a) The fitness development of test 'perfsyn_11'.



(b) The fitness development of test 'perfsyn_3'.

Figure 4.3: The fitness development of test 'perfsyn_3' and test 'perfsyn_11' when running WhiteSyn in *LoopFinding* mode with an input finding timeout of 5 minutes.

anymore until the timeout occurs. Since the path fitness weight ω (see Section 2.3) is set to 0.25 and the iteration fitness only decreases once the path fitness is 0, we can conclude that the path to the loop is found rather quickly (once the overall fitness reaches ~ 0.75), but achieving the required loop iterations, and therefore decreasing the iteration fitness, is hard.

Let us inspect the SUT of 'perfsyn_11', which is a single method and illustrated in Listing 4.2. The method under test `remove` belongs to the map implementation `ListOrderedMap` that maintains a list of keys to keep track of the insertion order of the elements. The targeted loop in `ArrayList` is called by `insertOrder.remove(key)` on `Line 5`. To fully iterate this loop μ times, the map needs to be filled with $\mu + 1$ different key-value pairs and the key added last needs to be provided as the argument to the method under test.

Random sampling of statements, as done by the genetic algorithm, is unlikely to create such a map. In contrast to many other tests, the involved data structure (`ListOrderedMap`) needs to be filled with *different* elements, rather than just being of a specific size. Additionally, specifically the *last* key in the map has to be provided as the argument to the method under test. $\mu = 16$ is still small enough, such that a test is generated, even tough, the fitness development noticeably flattens out towards lower fitness values. However, for $\mu \geq 64$, the chances of generating the required input within the timeout of 5 minutes are too low.

Figure 4.3b on the other hand, which depicts the fitness development of test 'perfsyn_3', shows a continuous improvement of the fitness for all values of μ until the fitness reaches 0. One can observe that even for $\mu = 1024$, it takes WhiteSyn only approximately 20 seconds to find the desired input. The method under test of 'perfsyn_3' does not require the elements in the corresponding data structure to be different for the targeted loop to be iterated μ times. Therefore, the genetic algorithm is much more likely to generate the required test. For 'perfsyn_3' in particular, it is enough to provide a `double` array of size $\mu + 1$ or larger and any `double` that is not in the array as argument to the method under test to fully iterate the targeted loop μ times.

Figure 4.3a shows that that the fitness development decreases much slower, if the number of loop iterations μ is increased. We can observe that increasing μ decreases the rate at which the fitness is decreased for other tests as well. There are two reasons for this behavior. First, the maximum allowed length of a generated program must be larger than μ (it is set to $\mu + 40$ in WhiteSyn) to allow tests that e.g. require μ `add` calls to a collection to be generated. Second, some parts of the genetic algorithm require linear runtime in regard to the program length, for example the validation of each statement in the generated program.

Both of these effects combined lead to less programs being generated per time if μ is increased, because each evolution step takes longer. However, this rate also depends on the test itself: if at least μ statements are required for the loop to be iterated μ times, the fitness decrease is much slower, as illustrated in Figure 4.3a, than if only a constant number of statements was required. In contrast, the fitness development depicted in 4.3b also shows a slower fitness decrease with larger values of μ, but the corresponding test 'perfsyn_3' does *not* require μ statements for its targeted loop to be triggered. Only two statements suffice, independent of μ. The fitness decrease for larger values of μ is still slower because some programs of size $\sim \mu$ are generated (because the maximum program length is $\mu + 40$), but these do not receive a low fitness value and therefore get quickly

removed from the population by the genetic algorithm. However, if larger programs receive better fitness values, the slower fitness decrease over time is amplified and the fitness decrease becomes much slower.

4.5.2 Integer Sampling

Another inherent drawback of genetic algorithms is apparent when sampling integers. If a randomly sampled integer is used as an initial size of a data structure or for an index-based access, chances are that the sampled index is invalid. The exact probability depends on the distribution used for sampling but for a Gaussian distribution with mean 0, as used by EvoSuite, there is a roughly 50% chance that a sampled integer is negative and e.g. initializing an array with a negative size or accessing an element at a negative index throws an exception. This leads to many generated tests that crash when executed and worsens the success rate when targeting SUTs that rely on sampled integers. Examples for this problem include test 'perfsyn_13' and test 'perfsyn_16' from Table 4.1. Adding symbolic execution and path constraints to the genetic algorithm might solve this issue.

4.5.3 Input Limitations

As shown in Table 4.1 and Table 4.2, many known bugs cannot be found by WhiteSyn by design, the most important groups being:

- Private or non-static inner classes
- No double loops
- Loops in native methods

With the current implementation of WhiteSyn, it is not possible to target private or non-static inner classes, since a generated input that uses these classes, cannot instantiate them without reflection. Of course, using such a class does not necessarily mean that it has to be instantiated from outside of the containing class. In fact, such an object is usually retrieved through a method of the containing class. For example, `ArrayList.listIterator` returns a `ListIterator` of dynamic type `ArrayList$ListItr`, which is a private inner class of `ArrayList`. However, this indirection is not supported by WhiteSyn.

Not targeting single loops was a deliberate design choice to focus on performance bugs that are amplified by loop nesting. In theory, it would be possible to also target single loops and analyze them with Toddler. However, this presumes that all paths to all loops are initially found during path finding, which is not feasible even for smaller libraries.

Finally, loops that are in native methods (e.g. `System.arrayCopy`) cannot be targeted, since WhiteSyn analyzes bytecode, which is by definition not available for native methods. Tools that rely on time measurements instead of static CFG analysis can still find such loops, because a potential redundancy manifests in a measurable increased runtime.

4.5.4 Input Size

During path finding, the main factor that causes the runtime to increase is the number of interface implementations. Since it is a good practice to declare variables with the type of the implementing interface (e.g. `List list = new ArrayList()`), at each call to such a variable, each possible receiver type (i.e. every implementation of the interface) has to be considered as a dynamic type of the receiver. Type analysis can statically bound the number of possible dynamic types (which WhiteSyn does), but usually has limited effects when dealing with variables that are not locals. As discussed in Section 3.7, we tried to prevent this issue with JDK collections, by restricting the classes and methods that are allowed to be used, both during path finding and input finding. However, different frameworks used in testing (e.g. Ant, ACC or Guava) introduce their own collections, again increasing the number classes and methods to be used.

As described in previous sections, if the genetic algorithm has many goals to cover at the same time, the input finding becomes less successful. On the one hand, this is caused by the inherent difficulty to cover multiple goals simultaneously, especially when they are independent or infeasible. On the other hand, the number of goals is usually correlated with the number of classes and methods in the SUT. To insert a statement into a program, the GA randomly samples a valid statement. Naturally, if there are more classes and methods available, the size of the set of available statements to sample from is increased as well. This might lead to many statements being inserted that do not contribute anything to covering the goal (e.g. `List.size()`) or even throw exceptions (e.g. `new ArrayList(-1)`).

4.5.5 Approach

As mentioned in Section 1, the first key contribution is the *approach* that is used by the presented prototype tool WhiteSyn. In particular, the static CFG analysis and the genetic algorithm for finding long-running tests are of interest. The evaluation has shown that a static CFG analysis is effective in finding loops that may cause long-running executions and might trigger performance bugs. The path finding algorithm (see Section 3.4.1) works well for most SUTs. However, targeting SUTs that have a large number of branches or interface implementations may lead to millions of paths, which negatively impacts the performance of the path finding algorithm. This is an inherent drawback of a static CFG analysis and exhaustively searching for all possible paths. The path finding algorithm's performance could be improved by incorporating symbolic execution and path constraints, which may reduce the number of paths that can be found by removing infeasible transitions from the CFGs.

Using a genetic algorithm has shown to be effective for generating tests with large input size. In contrast to tree-based approaches for input generation, genetic algorithms have a limited number of individuals in the population and therefore do not suffer from a search-space explosion. Additionally, the MOSA approach (see Section 3.5) facilitates generating tests for multiple goals at once, if the number of goals is not too large. However, as discussed in Section 4.5.1, an inherent disadvantage of a genetic algorithm is the difficulty of generating many statements that are not dependent on each other, e.g. add different elements to a data structure, instead of adding the same

element multiple times. Subsequently, genetic algorithms also do not perform well when aiming to generate complex input, such as an XML file. There are approaches that try to overcome this limitations with symbolic execution techniques.

5 Related Work

Automatic test generation for performance bugs is already done by existing tools. PerfSyn [27] uses a black box approach and a combinatorial search to generate tests that expose unexpected complexity or relative performance differences between two versions of the same method. WhiteSyn uses a white box approach and a genetic algorithm instead. Wise [5] generalizes results from small input sizes to generate tests that trigger worst-case complexity, whereas WhiteSyn focuses on finding redundant computations. Clarity [19] defines a class of performance bugs, named redundant traversal bugs, and implements a static analysis tool to automatically detect them. Travoli [21] has a similar but dynamic approach to help manually writing performance tests to find redundant traversal bugs. While redundant traversal bugs are similar to redundant computation bugs, WhiteSyn fully automatically generates tests without any manual effort. The Toddler [18] oracle used by WhiteSyn has been used for an automatic approach using patterns [7] to find loop inefficiencies. However, WhiteSyn does not require patterns or other input besides the SUT.

Previous work for performance test generation also deals with other types of performance bugs. EventBreak [26] generates performance tests for event-driven interfaces, rather than unit tests. SlowFuzz [24] detects algorithmic complexity vulnerabilities that can affect Web Application Firewalls or can be used to launch Denial-of-Service attacks. SpeedGun [25] focuses on changes in performance of thread-safe classes across different versions of the same class.

Certain tools that are designed for functional test generation are also used in various performance bugs detection tools e.g. to generate initial input. Randoop [20] incrementally builds functional tests by randomly generating statements and taking feedback from the execution into account. EvoSuite [8] generates tests that achieve high branch coverage. OCAT [10] aims to reduce the search space of object instances for test generation. Sushi [3] combines symbolic execution and search-based techniques to find complex test input.

Genetic programming was famously discussed by Koza in [12] and [13]. Generating a test whose execution trace follows a path through a CFG was proposed by [23]. This concept was later applied to unit tests by [28]. EvoSuite [8] also generates unit tests using a genetic algorithm, but targets high branch coverage by minimizing branch distances as defined in [16]. Unlike WhiteSyn, all of these approaches do not aim to expose performance bugs and their respective fitness functions do not take the number of achieved loop iterations into account. Based on EvoSuite, approaches for generating a whole suite of tests [9] or targeting many objectives at once with the MOSA approach [22] have been proposed. Recent work includes the MIO approach [2] that aims to further address the intrinsic properties of covering multiple goals at once, such as independence or infeasibility of goals.

6 Future Work

As the evaluation in Section 4.4 has shown, WhiteSyn does not perform well when using large SUTs as input, such as entire libraries, which is a drawback of our approach, as discussed in Section 4.5.5. While providing single methods as input mostly works well, the ability of WhiteSyn to generate tests that trigger long-running programs and, consequently, find redundancies, deteriorates rapidly when using larger SUTs. Researching this issue and improving WhiteSyn accordingly would greatly improve its usability and make it a viable tool for performance testing large systems. Additionally, the usability of WhiteSyn would benefit, if it could generate more complex input that exposes performance bugs, such as an XML file.

The evaluation of known performance bottlenecks and performance bugs in Section 4.1 and Section 4.2, respectively, has shown that for some tests, no input can be found, even with an increased input finding timeout. We observed that many of the failing tests are either caused by integer sampling (see Section 4.5.2) or targeted loops that are unlikely to be triggered (see Section 4.5.1). However, there might be additional reasons that cause the input finding to fail and further research is needed to analyze these limitations in detail.

As discussed in Section 4.5.3, WhiteSyn is not able to target non-static inner classes. Since many classes have parts of their implementations in such inner classes, this is a limitation that inherently restricts WhiteSyn's ability to find performance bugs. This drawback could be addressed by statically analyzing the relationship between targeted loops in inner classes and accessor methods (such as a getter) or using reflection.

WhiteSyn uses *LoopFinding* mode to generate tests that trigger many loop iterations that are then used by *RedundancyFinding* mode, because the Toddler oracle requires an input program for its analysis. However, the Toddler oracle could be replaced by other oracles that require an initial test for their analyses. For example, PerfSyn [27] defines an oracle that detects relative performance changes between two versions of a method and an oracle that detects unexpected asymptotic complexity of a method.

Lastly, WhiteSyn is currently not able to generate tests that trigger multiple iterations in loops with nesting level larger than 2. This limitation can be resolved by generalizing the approach discussed in Section 3.5.

7 Conclusion

We have presented a white box approach for test generation using static CFG analysis and a genetic algorithm and the prototype tool WhiteSyn that implements this approach. The approach proved to be powerful for generating tests that trigger many loop iterations and expose redundant computations, but has some drawbacks, e.g. handling large codebases and generating complex input.

The implemented prototype tool WhiteSyn is a fully automatic test generation tool that aims to generate long-running tests and find and expose performance bugs in a system under test. WhiteSyn is inspired by PerfSyn, but uses a white box (instead of black box) analysis and a genetic algorithm (instead of a combinatorial search), as defined by our approach. WhiteSyn statically finds paths that are of interest (e.g. reach a certain loop depth) and uses the genetic algorithm to find input that triggers them. Given the triggered path of a candidate solution, a custom fitness function that takes the paths of interest and a minimal number of loop iterations into account, is used to guide the evolution of the genetic algorithm.

As a proof of concept, we implemented the Toddler oracle for finding redundant computations in nested loops, which presumes input that iterates possibly redundant, nested loops. This input can be generated by WhiteSyn itself.

Testing has shown that our approach is most effective when targeting single methods. WhiteSyn is able to consistently generate tests that trigger up to 32 loop iterations and for some tests even reliably achieves up to 256 loop iterations. WhiteSyn can expose many performance bugs, namely redundant computations, by automatically generating corresponding tests.

Bibliography

[1] F. E. Allen and F. E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, July 1970.

[2] A. Arcuri. Many Independent Objective (MIO) Algorithm for Test Suite Generation. In T. Menzies and J. Petke, editors, *Search Based Software Engineering*, Lecture Notes in Computer Science, pages 3–17. Springer International Publishing, 2017.

[3] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè. Combining Symbolic Execution and Search-based Testing for Programs with Complex Heap Inputs. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 90–101, New York, NY, USA, 2017. ACM.

[4] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and Extensible Component Systems*, 2002.

[5] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *2009 IEEE 31st International Conference on Software Engineering*, pages 463–473, May 2009.

[6] K. Deb. Multi-Objective Optimization. In E. K. Burke and G. Kendall, editors, *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, pages 273–316. Springer US, Boston, MA, 2005.

[7] M. Dhok and M. K. Ramanathan. Directed Test Generation to Detect Loop Inefficiencies. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 895–907, New York, NY, USA, 2016. ACM.

[8] G. Fraser and A. Arcuri. Evolutionary Generation of Whole Test Suites. In *2011 11th International Conference on Quality Software*, pages 31–40, July 2011.

[9] G. Fraser and A. Arcuri. Whole Test Suite Generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, Feb. 2013.

[10] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. OCAT: Object Capture-based Automated Testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 159–170, New York, NY, USA, 2010. ACM.

[11] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[12] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[13] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA, 1994.

[14] P. Leitner and C.-P. Bezemer. An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 373–384, New York, NY, USA, 2017. ACM.

[15] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1013–1024, New York, NY, USA, 2014. ACM.

[16] P. McMinn. *Search-Based Software Test Data Generation: A Survey*. 2004.

[17] A. Nistor, P. Chang, C. Radoi, and S. Lu. CARAMEL: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 902–912, May 2015.

[18] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571, Piscataway, NJ, USA, 2013. IEEE Press.

[19] O. Olivo, I. Dillig, and C. Lin. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 369–378, New York, NY, USA, 2015. ACM.

[20] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84, May 2007.

[21] R. Padhye and K. Sen. Travioli: A Dynamic Analysis for Detecting Data-structure Traversals. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 473–483, Piscataway, NJ, USA, 2017. IEEE Press.

[22] A. Panichella, F. M. Kifetew, and P. Tonella. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, Apr. 2015.

[23] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-Data Generation Using Genetic Algorithms. *Software Testing, Verification And Reliability*, 9:263–282, 1999.

[24] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2155–2168, New York, NY, USA, 2017. ACM.

[25] M. Pradel, M. Huggler, and T. R. Gross. Performance Regression Testing of Concurrent Classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 13–25, New York, NY, USA, 2014. ACM.

[26] M. Pradel, P. Schuh, G. Necula, and K. Sen. EventBreak: Analyzing the Responsiveness of User Interfaces Through Performance-guided Test Generation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 33–47, New York, NY, USA, 2014. ACM.

[27] L. D. Toffola, M. Pradel, and T. R. Gross. Synthesizing Programs That Expose Performance Bottlenecks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 314–326, New York, NY, USA, 2018. ACM.

[28] P. Tonella and P. Tonella. Evolutionary testing of classes. *ACM SIGSOFT Software Engineering Notes*, 29(4):119–128, Nov. 2004.

[29] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–, Mississauga, Ontario, Canada, 1999. IBM Press.

[30] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, June 1994.