Evaluating CAVM: A New Search-Based Test Data Generation Tool for C

Author information omitted

for double-blind review

Abstract. We present CAVM (pronounced "ka-boom"), a new search-based test data generation tool for C. CAVM is developed to augment an existing commercial tool, XYZXYZ (hidden for double blind), which uses static analysis and input partitioning to generate test data. Unlike the current state-of-the-art search-based test data generation tool for C, Austin, CAVM handles dynamic data structures using purely search based techniques. We compare CAVM against XYZXYZ and Austin using 49 C functions, ranging from small anti-pattern case studies to real world open source code and commercial code. The results show that CAVM can cover branches that neither XYZXYZ nor Austin can, while also exclusively achieving the highest branch coverage for 20 of the studied functions.

1 Introduction

Search-based automatic test data generation has been of interest to researchers for several decades [4, 12, 16, 17, 19, 21, 25], yet there is little evidence that this research has made the transition into industrial practice. Furthermore, most recent research developments have centred on test data generation for object-oriented code written in Java [4–6], to the neglect of other languages in prevalent use, for example C, which is still the second most popular language according to ratings websites (e.g., [1]).

In this paper, we find that there are challenges for search-based test data generation for C programs that remain unsolved. These challenges primarily relate to the generation of inputs consisting of dynamic data structures. In order to test C functions with pointer inputs, a data structure has to be found that is of the right "shape". The same problem does not exist for Java programs, because in general, the required structures of objects needed to adequately test a method can be generated by using the constructor and method calls that exist the code base under test. Solving these challenges is of high importance: As this paper evidences, industry-strength automated test data generators for C are still lacking, while robust techniques are required for other fields of research whose tools focus on the C language — for example automatic program repair [7, 26], automatic fault localization [14, 27], and genetic improvement [2, 10]. Many of these applications either adopt multi-objective optimisation or concern non-functional properties of the target software system, which are more easily dynamically optimized and incorporated into search-based test data generators rather than those that rely on full or partial static analysis (such as Dynamic Symbolic Execution [3]).

Our analysis begins with two existing tools for generating inputs to test functions written in the `C` language: `XYZXYZ`, an industrial tool; and a search-based test data generation tool, `Austin`. `XYZXYZ` adopts a simple yet effective heuristic to generate inputs consisting of dynamic data structures. `Austin`, in contrast, performs a lightweight symbolic analysis to generate the shape of the dynamic data structure, which is subsequently filled using the Alternating Variable Method (AVM) [20]. Although, hitherto, `Austin` represents the current state of the art for search-based test data generation for `C` code, it is no longer actively maintained [18].

We therefore introduce and evaluate `CAVM` (pronounced "ka-boom"), a new search-based test data generation tool for `C`. `CAVM` is also based on the AVM, but differs from `Austin` in that it generates inputs consisting of dynamic data structures using purely a search-based technique: *growing* the appropriate shape of the dynamic data structure, as well as filling it with data, is part of the metaheuristic search performed. It also supports generation of string input (i.e., `char` array) for `strcmp` using code rewriting.

We compare `CAVM` against `XYZXYZ` and `Austin` with respect to their relative effectiveness for `C` code involving dynamic data structures. The empirical evaluation studies small anti-pattern case studies, which are known to be challenging for `XYZXYZ`, as well as real world open source and commercial code. The results show that our new algorithms implemented into `CAVM` can cover branches that neither `XYZXYZ` nor `Austin` can.

The contributions of this paper are therefore as follows:

1. A new search-based algorithm for generating test inputs consisting of dynamic data structures for functions written in the `C` language, implemented into a test data generation tool called `CAVM` (Section 3).
2. An empirical study that compares `CAVM`, `XYZXYZ`, and `Austin` with respect to their ability to generate test input consisting of dynamic data structures, using 49 `C` functions ranging from small anti-pattern case studies to modules in commercial `C++` front-end (Section 4).
3. Results that illustrate types of branches for which `CAVM` can generate test data where neither the existing state-of-the-art tool, `Austin`, nor the commercial tool, `XYZXYZ`, cannot (Section 5).

We begin by detailing background important to the `XYZXYZ` and `Austin` tools.

2 Background

In this section, we introduce two tools for generating test data for `C` functions. The first, `XYZXYZ`, is a commercial tool that derives inputs through static analysis. The second, `Austin`, combines dynamic search for numeric inputs while deriving information about the required shape of dynamic data structures through a lightweight symbolic execution routine. Until now, `Austin` has represented the state of the art in terms of search-based test data generation for `C` code.

```
1 void foo() { return 10; }
2
3 void testMe(int a, int b, int c) {
4   if(a < 42)
5     if(b == c)
6       if(c == foo())
7         // target
8 }
```

(a) Example with numeric inputs

```
1  typedef struct _data {
2    int* internal;
3    int a, b;
4    struct _data* next;
5  } Data;
6
7  void testMe(Data *d) {
8    if (d != NULL) {
9      Data *d_in;
10     d_in = d->next;
11     if (d_in != NULL)
12       if (d_in->a == 2)
13         // target
14   }
15 }
```

(b) Example with a pointer input

Fig. 1: Example code for explaining the operation of different approaches to automatic test data generation

2.1 XYZXYZ

`XYZXYZ`, developed by XYZ Technology Inc., is a commercial test data generation tool for `C` and `C++`. It is used by a variety of industry clients to test and obtain structural coverage for mission critical software systems in the automotive, defence, and aerospace domains[1]. `XYZXYZ` adopts a number of techniques including periodic value generation, random generation, and pairwise input partitioning. The experiences of `XYZXYZ` engineers with respect to their clients suggest that the pairwise partitioning is the most effective method.

Let us use the example of Figure 1a to illustrate how the pairwise input partitioning of `XYZXYZ` works, as well as its weaknesses. It adopts lightweight static analysis to identify boundary values and the resulting input partitions. For branches with fixed concrete boundary values, such as the one in Line 4 in Figure 1a, `XYZXYZ` can generate values for the variable `a` efficiently. For example, given the predicate `a < 42`, it will try, for `a`, 41, 42, 43 (the boundary value as well as its neighbours), values from the intervals (i.e., values that are sufficiently higher and lower than 42), as well as so called *type partitions*: the minimum, the maximum, and the median value for the given primitive type.

Subsequently, `XYZXYZ` will attempt to achieve pairwise interaction coverage, similar to those used in Combinatorial Interaction Testing (CIT) [23], between possible values for each variable. Note that this process does not involve any actual execution of the target function: most of the execution time is consumed by the static analysis. While effective for certain branches, the pairwise approach of `XYZXYZ` does have weaknesses. For example, if the boundary is set by parameters (Line 5 of Figure 1a) or function calls (Line 6 of Figure 1a), the dynamic boundary values elude the static analysis, resulting in low branch coverage.

`XYZXYZ` adopts a simple yet effective heuristic for generating dynamic data structures. First, pointers are regarded as an array containing a single item: if the item is of a composite type (e.g., `struct`), the values of its members that are used in the target function are populated using the pairwise method; others are assigned with `0`. Second, if the composite type includes a recursive pointer

[1] Please refer to [redacted for double blind review]

(i.e., a pointer that points to its own type), the pointer is instantiated by depth of 1, but *not* its members. For example, given the code snippet in Figure 1b, `XYZXYZ` will instantiate the parameter `d` and `d->next`, but it will not assign any values to the `d->next->a` variable. This is to avoid the explosion of the number of test cases due to the pairwise combinations.

2.2 Austin

`Austin` is a search-based test data generation tool for `C` that incorporates the use of dynamic symbolic evaluation techniques and constraint solving. Values for numerical inputs are sought using the AVM, while pointer inputs are assigned by "solving" a pointer equivalence graph (inspired by a similar mechanism used by CUTE [24]).

`Austin` utilises these approaches as follows. Given a `C` function under test, `Austin` selects an uncovered branch as the "target". Given an initial, randomly-generated input, `Austin`'s instrumentation detects whether the branch was covered, or whether execution flow diverged from the target. If execution flow diverged from the target, then the program evaluated a condition at some "critical" decision statement in the function that led to the target branch being missed. This decision statement corresponds to a node in the control flow graph of the function on which the target branch is control dependent, for example an "`if`" statement in which the branch is nested (e.g., the false branches from the decisions at lines 8, 11 and 12 for Figure 1b and the target true branch from line 12).

At this point, `Austin` invokes its dynamic symbolic execution engine to derive a path condition, a constraint over the inputs of the function describing when a path through the function will be executed. The path condition is derived for the path executed by the current input up to and including the critical decision node. The condition corresponding to that of the critical node is then inverted, so that the overall path condition now describes the path that needs to be taken to "correct" execution flow so that the target is reachable.

`Austin` analyses the path condition to decide whether it needs to invoke the AVM to find numerical inputs to the function, or whether the problem is a result of pointer inputs. If the latter, `Austin` works to create the aforementioned pointer equivalence graph. The path condition is simplified to refer to pointer constraints of the form $x = y$ and $x \neq y$, where x and y may be `NULL` or a symbolic variable corresponding to a pointer input. `Austin` derives a pointer equivalence graph from the path condition by grouping pointers that are equivalent to one another into nodes of the graph, with edges added between nodes to capture non-equivalence. This graph is then "solved" to generate pointer inputs as follows. For each node n, if n does not have an edge to the node containing "`NULL`", `Austin` sets all concrete pointer inputs represented by the symbolic variables represented by the node to `NULL`. If, however, n represents the address of another symbolic variable s, `Austin` assigns all concrete pointer inputs to the location pointed at by s. Otherwise, `Austin` creates a new memory location, assigning all concrete pointer inputs in n to that location.

For example, in Figure 1b the target is the true branch from line 12. Although the branching condition involves satisfying a numerical constraint, there is a

dynamic data structure involved that must be initialised correctly. `Austin` infers three nodes in the pointer equivalence graph, one involving `d` (n_1), one involving both `d->next` (n_2) and one for `NULL` (n_3). There is an edge between n_3 and n_1, inferred from the branching condition on line 8 that must be executed as true, since `d` must not be `NULL`. The results of the symbolic execution also reveal that the assignment of `d->next` to `d_in` on line 10 — and the subsequent need to execute line 11 as true — mean that there is also an edge between n_3 and n_2. The result of solving this graph therefore means that `d` is initialised to a new memory location, while `d->next` is also initialised to a separate location. The AVM search then solves the condition on line 12 that `d_in->a` (i.e., `d->next->a`) must be set to `2`.

`Austin` is no longer actively maintained (the last commit to its open source repository was in 2011 [18]) and fails to cover certain types of branches — as we show in Section 5 — therefore motivating a new tool, which we introduce next.

3 `CAVM`: A New `C` Test Data Generation Tool

`CAVM` is an open source byproduct of an industry collaboration, the aim of which is to augment `XYZXYZ` with a search-based software testing technique so that it can deal with challenging branches more effectively.

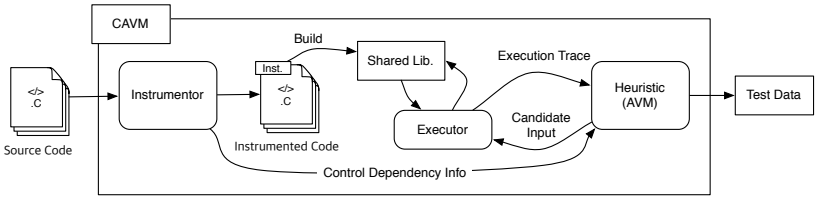

Fig. 2: Overall Architecture of `CAVM`

3.1 Overall Architecture

The overall architecture of `CAVM` is depicted in Figure 2. First, `CAVM` starts by instrumenting the given source code. Instrumentation is performed directly to the source code in `C` through `Clang`. After replacing the original source code with the instrumented version, the target program is built as a shared library, so that `CAVM` can import and call the target function directly without having to add an extra entry point. Subsequently, `CAVM` applies an extended version of AVM to the target program to search for the test input that covers a specific branch.

3.2 Extending the AVM for Dynamic Data Structures

Algorithm 1 presents the pseudocode for the search algorithm used by `CAVM`. For the sake of brevity, the pseudocode assumes that whenever *fitness* reaches

Algorithm 1: Local Search Algorithm of `CAVM`

LocalSearch($\overrightarrow{x}$, *current*)
Input: An input vector, $\overrightarrow{x}$, and the current search target, *current*
Output: The minimum fitness value found, $fitness$, and the input vector that corresponds to the value, $\overrightarrow{x}$

```
(1)   while budget > 0 and fitness > 0
(2)     e = NEXT(x⃗, current)
(3)     if IsPrimitive(e)
(4)       fitness ← IterativePatternSearch(x⃗, e)
(5)     else if IsStruct(e)
(6)       fitness ← LocalSearch(x⃗, e.members)
(7)     else if IsPointerToPrimitives(e)
(8)       foreach i in e
(9)         fitness ← IterativePatternSearch(x⃗, i)
(10)      e′ ← Grow(e)
(11)      x⃗ ← x⃗(e ← e′)
(12)      fitness ← Evaluate(x⃗)
(13)    else if IsPointerToStruct(e)
(14)      if e is NULL
(15)        e′ ← instantiate(e)
(16)        x⃗ ← x⃗(e ← e′)
(17)        fitness ← Evaluate(x⃗)
(18)      else
(19)        fitness ← LocalSearch(x⃗, *e)
(20)  return x⃗, fitness
```

the optimum, the control flow breaks from the main loop. We also leave out the check for whether fitness improved during the last full iteration over elements of the input vector $\overrightarrow{x}$. For primitive data types, line 4 invokes Iterated Pattern Search (IPS) [15, 20]. The remaining cases deal with dynamic data structures. Note that `CAVM` initialises all pointers in the input vector $\overrightarrow{x}$ to `NULL`.

Lines 5 and 6 handle `struct` by flattening the members of the current `struct` elements in $\overrightarrow{x}$, recursively when presented with nested `struct` elements. Lines 7 to 19 handle pointers. `CAVM` considers pointers to primitive types as arrays: Lines 7 to 12 apply IPS to each element of the current array, and grows the array by one if the search does not finish.

Finally, lines 13 to 19 deals with pointers to `struct`. If the pointer is currently `NULL`, `CAVM` checks if instantiating the current pointer (line 15) improves the fitness (primitive members are randomly initialised). If not, `CAVM` tries to search for the values of `*e`, that is, the members of the pointed `struct` (lines 18 and 19).

Let us consider the code in Figure 1b as an example. `CAVM` starts with the input `d` being `NULL`. Since this input does not reach the target, `CAVM` instantiates `d`, which will cover the true branch from line 8. Subsequently, `CAVM` will apply IPS to `d->a` and `d->b` but without improving the fitness. Eventually `CAVM` instantiates

`d->internal` as well as `d->next`: while applying IPS again to primitive elements, `CAVM` will change the value of `d->next->a` to 2 and reach the target.

3.3 String Comparison

Since strings are represented as an array of characters in the `C` language, `CAVM` treats searching for a string test input as searching for a specific character array. While `CAVM` can search for primitive arrays with specific contents, the use of `strcmp` presents an additional challenge because its return value does not easily translate to fitness value that provides good guidance: `strcmp` returns a single integer, representing whether the two input strings are identical, or one is alphabetically *higher* (i.e., is after the other in alphabetical order) than the other. The lack of "distance" information between two strings makes it hard for `CAVM` to determine the search direction.

To provide string distances, `CAVM`'s instrumentation process replaces usages of `strcmp` in predicates with `strcmp2`, which is our wrapper for `strcmp`. This is essentially the same approach as Fraser and Arcuri [4], which replaces the `String .equals()` method in Java. However, none of the existing search-based test data generation tools for `C` implement this idea.

The `strcmp2` returns signed string distances instead of signed integers: the string distance is defined as `lengthDiff` + $norm$(`charDiff`). The `lengthDiff` is the difference in length between two input strings, whereas the `charDiff` is the sum of character distances between two input strings, up to the shorter length. We use the standard $norm(x) = 1 - 1.001^{-x}$ for normalisation [11].

4 Experimental Setup

4.1 Research Questions

We seek to answer the following research questions with the empirical study.

- **RQ1. Effectiveness:** Among `XYZXYZ`, `Austin`, and `CAVM`, which tool achieves the highest branch coverage against the studied target subjects?
- **RQ2. Efficiency:** Among `XYZXYZ`, `Austin`, and `CAVM`, which tool achieves branch coverage the most efficiently?

We answer RQ1 and RQ2 by applying all studied tools to the target subject functions in `C`. Since `Austin` and `CAVM` adopt stochastic approaches, we will report the average coverage for RQ1 over 20 runs, the average number of fitness evaluations, as well as the average wall clock time required for test data generation for RQ2 over 20 runs. We only evaluate the deterministic heuristic of `XYZXYZ`, and therefore we do not repeat runs with this tool.

4.2 Subjects

Table 1 contains the list of subject functions that we study in this paper: they are from a variety of sources, ranging from small toy examples to a commercial `C++` frontend. The anti-pattern subject is a set of branches that `XYZXYZ` is known to be unable to cover: these are the minimum working examples that contain only the problematic structural patterns. `Line`, `Calendar`, `Triangle`, and `AllZeros` examples are ported to `C` from McMinn and Kapfhammer [20] and constitute the baseline examples. `LinkedList` is a collection of utility function implementations for the singly linked list in `C`, taken from an on-line tutorial, whereas `BinaryTree` contains seven functions from the textbook by Horowitz et al. [13]. Finally, `busybox-ls` contains five functions from the open source implementation of `ls` utility for the `busybox` package, whereas `decode.c` contains 24 functions chosen from a name demangler module for `C++` frontend, developed by the Edison Design Group. In total, we study 482 branches in 49 functions.

Table 1: Subject `C` Functions Studied

Subject	Description	Branches	*	Rec. *	struct	strcmp
`AllZeros`		6	✓	-	-	-
`Calendar`	Examples from `AVMf` [20]	46	-	-	-	-
`Line`		14	-	-	✓	-
`Triangle`		16	-	-	-	-
`XYZXYZ` Antipatterns	Set of branches that `XYZXYZ` cannot cover	16	✓	✓	✓	✓
`LinkedList`	5 utility functions for singly linked list[2]	26	✓	✓	✓	-
`BinaryTree`	7 tree-related functions from a textbook by Horowitz et al. [13]	30	✓	✓	✓	-
`busybox-ls`	5 functions from `ls` in `Busybox 1.2.0`[3]	32	✓	-	-	-
`decode.c`	22 functions from `decode.c`[4]	296	✓	-	✓	-
Total	49 `C` functions	482				

The column denoted by "`*`" shows whether any function contains a pointer type parameter, whereas the column "Rec. `*`" shows whether some of those contain recursive data structures (e.g., a pointer to a `struct`, which in turn contain a pointer to its own type). Similarly, the column `struct` shows whether any parameter is of a `C` structure type (`struct`), and the column `strcmp` shows whether any input parameter is being compared as a string using `strcmp`.

Note that the intention behind choosing these functions is actually not to perform an empirical study based on an unbiased sample of arbitrary `C` functions in the wild. Rather, we wanted to make a focused selection of functions, around the particular features and usages of `C` language, against which we compare the

[2] Taken from an on-line tutorial: `http://milvus.tistory.com/17`

[3] BusyBox is a collection of common UNIX utilities in a single small executable: `https://busybox.net`.

[4] `https://www.edg.com/c`

studied tools: namely, the use of pointers, arrays, and dynamic data structures, as well as the specific current weaknesses of XYZXYZ. Source code for all studied subjects are available at: [redacted for double blind].

4.3 Configurations

While CAVM allows the user to set the search range for each input parameter of the target function, Austin lacks such control. Consequently, we do not narrow down the input range and use the default range for each primitive type, so that both tools search in the same space. For both Austin and CAVM, we set the maximum number of fitness evaluations for each target branch to 1,000, and the timeout duration for each target function to five minutes. Note that both tools collect "collateral" coverage [8] (i.e., coverage of branches that are not the target but nonetheless covered by a test case generated by a tool[5]). Any collateral coverage achieved within five minutes counts in the final results. However, if a tool does not terminate within the five minute timeout, we record 0% coverage.

XYZXYZ implements multiple test input generation techniques, but we only use the pairwise approach because it is known to be the most effective, as discussed in Section 2.1. Note that, unlike Austin and CAVM, XYZXYZ does not require any actual execution of the target function in order to generate the test data, due to the way its heuristic works (see Section 2.1). Furthermore, because XYZXYZ can only be operated via GUI, direct comparison of execution time is infeasible. Consequently, we do not report the execution time of XYZXYZ: however, we do report approximate wall clock execution time for representative cases.

Since Austin relies on the C Intermediate Language (CIL) [22], the number of branches Austin attempts to cover can be different from those for XYZXYZ and CAVM. For example, Austin breaks down composite predicates into nested if statements. We therefore manually inspected the results of Austin for the affected functions and matched the branches attempted with those for XYZXYZ and CAVM.

4.4 Environments

CAVM is written in C/C++ as well as Python. The target code instrumentation is written in C/C++ and depends on clang version 3.9.0 and GNU gcc version 4.9 or higher. The AVM search is written in Python 3 and depends on CFFI[6] as well as Python runtime version 3.5 or higher.

For the experiment, CAVM is executed on a machine with Intel Core i7-6700K 4.0GHz and 32GB RAM running Ubuntu 14.04 LTS. Due to specific dependencies, Austin is executed on the same machine running Ubuntu 12.04.5 LTS. XYZXYZ is executed on a machine with Intel Core i5-6600 3.9GHz and 16GB RAM running Windows 7. We allow the different hardware environment because we do not compare the execution time directly for the reasons described in Section 4.3.

[5] Here, we define collateral coverage as branches that are covered in addition to the original target by the final, generated test cases.

[6] C Foreign Function Interface: http://cffi.readthedocs.io

Table 2: Average branch coverage (μ) and standard deviation (σ) from `XYZXYZ`, `Austin`, and `CAVM` over 20 runs: the highest coverage for each function is typeset in bold. `Br.` contains number of branches; `XY` stands for `XYZXYZ`.

Function	Br.	XY	Austin		CAVM	
			μ	σ	μ	σ
AVMf						
allzeros$^{\square}$	6	0.00	0.00	0.00	**83.33**	0.00
calendar*	46	**100.00**	0.00	0.00	0.00	0.00
Antipatterns						
case1	4	0.00	**100.00**	0.00	**100.00**	0.00
case2	4	75.00	**100.00**	0.00	**100.00**	0.00
case3	2	50.00	**100.00**	0.00	**100.00**	0.00
case4§	2	0.00	0.00	0.00	**100.00**	0.00
case5	2	50.00	**100.00**	0.00	**100.00**	0.00
case6	2	50.00	**100.00**	0.00	**100.00**	0.00
LinkedList						
delete$^{\diamond}$	6	**100.00**	**100.00**	0.00	16.67	0.00
insert$^{\diamond}$	8	87.50	**100.00**	0.00	50.00	0.00
modify$^{\diamond}$	4	75.00	**100.00**	0.00	38.75	12.76
print_list	2	**100.00**	**100.00**	0.00	**100.00**	0.00
search	6	**100.00**	0.00	0.00	**100.00**	0.00
busybox-ls						
bold	2	50.00	**100.00**	0.00	**100.00**	0.00
dnalloc	2	**100.00**	**100.00**	0.00	**100.00**	0.00
fgcolor	2	**100.00**	**100.00**	0.00	**100.00**	0.00
my_stat	10	0.00	0.00	0.00	0.00	0.00
scan_one_dir	16	**6.25**	0.00	0.00	0.00	0.00
BinaryTree						
inorder	2	**100.00**	**100.00**	0.00	**100.00**	0.00
iter_inorder	4	0.00	0.00	0.00	**100.00**	0.00
iter_search	6	**100.00**	0.00	0.00	**100.00**	0.00
level_order	8	62.50	0.00	0.00	**100.00**	0.00
postorder	2	50.00	**100.00**	0.00	**100.00**	0.00
preorder	2	50.00	**100.00**	0.00	**100.00**	0.00
search	6	**100.00**	0.00	0.00	**100.00**	0.00

Function	Br.	XY	Austin		CAVM	
			μ	σ	μ	σ
AVMf						
line†	14	**100.00**	0.00	0.00	28.57	0.00
triangle‡	16	**93.75**	0.00	0.00	89.06	5.32
decode.c						
func1	2	**100.00**	0.00	0.00	**100.00**	0.00
func2	2	**100.00**	0.00	0.00	**100.00**	0.00
func3	48	10.42	0.00	0.00	**29.90**	5.63
func4	14	21.43	0.00	0.00	**71.07**	6.34
func5	14	**21.43**	0.00	0.00	0.00	0.00
func6	16	18.75	0.00	0.00	**27.14**	9.44
func7	30	6.67	0.00	0.00	**11.56**	1.79
func8	6	50.00	0.00	0.00	**75.83**	12.65
func9	44	4.55	0.00	0.00	**69.66**	7.31
func10	28	7.14	0.00	0.00	**62.32**	10.20
func11	2	**100.00**	0.00	0.00	**100.00**	0.00
func12	4	25.00	0.00	0.00	**27.50**	7.69
func13	4	50.00	0.00	0.00	**73.75**	5.59
func14	2	50.00	0.00	0.00	**52.50**	11.18
func15	2	50.00	0.00	0.00	**97.50**	11.18
func16	12	8.33	0.00	0.00	**22.50**	18.56
func17	4	25.00	0.00	0.00	**27.50**	11.18
func18	4	50.00	0.00	0.00	**64.17**	6.11
func19	28	3.57	0.00	0.00	**8.75**	3.57
func20	8	87.50	0.00	0.00	**100.00**	0.00
func21	4	**100.00**	0.00	0.00	**100.00**	0.00
func22	18	**100.00**	0.00	0.00	**100.00**	0.00

Section 5.1 discusses the following issues.
$\square$: indirect dependency. $*$: large search space.
†: low success rates. ‡: infeasible branches.
$\diamond$: imprecise dependency analysis. §: `strcmp`.

5 Results

5.1 Effectiveness

Table 2 contains the coverage results from 20 repetitive runs of `Austin` and `CAVM`, as well as the single run of `XYZXYZ`. Note that the functions in `decode.c` have been renamed in the table to save space: their full names, as well as their source code and the box plots of the coverage results will be available from the accompanying web page. For `Austin` and `CAVM`, we report mean (μ) and standard deviation (σ): the highest coverage is typeset in bold. Out of 49 functions, there are 5 functions for which `XYZXYZ` alone achieves the highest branch coverage, and two functions for which `Austin` does the same. `CAVM` alone achieves the highest branch coverage for 20 functions. Notably, `Austin` fails to cover any branch of functions in `decode.c` within five minutes.

We manually analysed the hard-to-cover branches in the smaller benchmarks and identified the following common issues (each issue can be cross-referenced to Table 2 through the symbols):

– Indirect control dependency (□): one of the branches in the `allzeros` function requires the number of zeros in the input array to be equal to the size of input: `CAVM` fails to cover this branch. `CAVM` does not receive any guidance through the fitness function because the counter for the number of zeros is changed in another branch that does not depend on the target branch, similar to the flag problem [9]. This results in `CAVM` repeating random restarts.
– Large search spaces (∗): a `for` loop in `calendar` consumes a large amount of time when inputs are initialised from a large range. Since the loop iterates over the range between two integer inputs, the number of iterations can be up to the range of integers in `C`. This leads to frequent timeouts and, consequently, 0% coverage. When the input variable range is set to [-100, 100], `CAVM` consistently achieves 100% coverage.
– Low success rate (†): some branches in the `line` function are simply hard to cover under the given timeout and evaluation budget. While `CAVM` sometimes succeeds to cover all branches in `line`, the average coverage suffers from runs that failed to cover the hard branches.
– Infeasible branches (‡): the function `triangle` contains an infeasible branch. Consider the following code snippet from `triangle`:

```
if(a == b) { ... } else { if(a == b) { ... }}
```

The true branch of second predicate is logically infeasible because of the first one. Apart from this branch, `CAVM` and `XYZXYZ` cover all branches in `triangle`.
– Use of `strcmp` (§): `case4` in `Antipatterns` contains a call to `strcmp`, which neither `XYZXYZ` nor `Austin` supports.
– Imprecise control dependency analysis (◊): currently `CAVM` suffers from imprecise control dependency analysis; it cannot detect implicit control dependencies between branches caused by, for example, a `return` in the middle of a function. Consider the following code snippet:

```
if(x > 42) return; if(y == 7)...
```

Both the true and the false branch of the second `if` statement depend on the false branch of the first one. However, this dependency is implicit, i.e. it is not expressed in the nested structure. The current control dependency analysis of `CAVM` fails to capture this. Consequently, `CAVM` cannot compute the fitness values correctly for these branches and cannot cover them. When we manually made the control dependency explicit (by inserting `else` appropriately), `CAVM` achieves an average of approximately 60% branch coverage for functions `delete`, `insert`, and `modify` in the `LinkedList` subject, with some individual runs achieving 100% coverage. Precise control dependency analysis for the full set of structural constructs of `C` is an item of future work.

Finally, let us discuss the performance of `Austin`. `Austin` requires an explicit pointer constraint in the source code of the target function in order to instantiate any pointer. If the code does not compare a given pointer to `NULL`, the pointer will not be instantiated. After confirming this behaviour to be intended with the main

developer of `Austin`, we inserted explicit `NULL` checks to smaller benchmarks (`Antipatterns`, `AVMf`, `LinkedList`, and `BinaryTree`), but opted not to modify the real world subjects (`ls` and `decode.c`). This results in the consistent 0% coverage for functions in `decode.c`, as they all require pointer parameters.

Based on the results in Table 2, we answer RQ1: `CAVM` can cover branches that neither `XYZXYZ` nor `Austin` can. In particular, `Austin` has a significant limitation regarding pointer instantiation.

Table 3: Average execution time (μ) in seconds and standard deviation (σ) required by `Austin` and `CAVM` over 20 runs: the shortest time for each function is denoted in bold, whereas timeouts are marked with a dash.

Function	Austin		CAVM		Function	Austin		CAVM	
	μ	σ	μ	σ		μ	σ	μ	σ
AVMf					**AVMf**				
allzeros$^{\square}$	-	-	**3.62**	0.34	line†	-	-	**43.75**	1.16
calendar*	-	-	-	6.66	triangle‡	-	-	**12.04**	2.28
Antipatterns					**decode.c**				
case1	2.01	1.20	**1.30**	0.58	func1	-	-	**0.98**	0.02
case2	1.85	0.20	**0.43**	0.02	func2	-	-	**0.59**	0.01
case3	0.76	0.11	**0.74**	0.18	func3	-	-	**196.48**	17.38
case4	-	-	**0.75**	0.11	func4	-	-	**16.61**	3.01
case5	**0.72**	0.19	0.79	0.19	func5	-	-	-	-
case6	**0.99**	0.17	1.18	0.02	func6	-	-	**41.46**	5.69
LinkedList					func7	-	-	**150.08**	4.83
delete$^{\diamond}$	**2.44**	0.79	24.68	4.71	func8	-	-	**8.18**	1.31
insert$^{\diamond}$	**2.55**	0.87	17.11	0.36	func9	-	-	**77.69**	15.59
modify$^{\diamond}$	**1.81**	0.26	13.67	4.93	func10	-	-	**54.38**	10.44
print_list	**0.15**	0.01	0.33	0.01	func11	-	-	**0.67**	0.06
search	-	-	**1.02**	0.19	func12	-	-	**11.21**	1.38
busybox-ls					func13	-	-	**5.68**	0.97
bold	0.88	0.89	**0.31**	0.15	func14	-	-	**3.82**	0.81
dnalloc	**32.31**	17.54	62.82	37.86	func15	-	-	**2.05**	0.90
fgcolor	1.01	0.75	**0.36**	0.22	func16	-	-	**37.74**	9.25
my_stat	-	-	-	-	func17	-	-	**10.98**	1.66
scan_one_dir	-	-	-	-	func18	-	-	**9.99**	1.21
BinaryTree					func19	-	-	**123.74**	6.86
inorder	**0.14**	0.02	0.19	0.01	func20	-	-	**1.51**	0.06
iter_inorder	-	-	**71.08**	0.34	func21	-	-	**0.73**	0.01
iter_search	-	-	**0.86**	0.22	func22	-	-	**12.07**	0.71
level_order	-	-	**143.82**	0.49	Section 5.2 discusses the following:				
postorder	**0.13**	0.01	0.19	0.00	$\square$: indirect dependency. $*$: large search space.				
preorder	**0.14**	0.01	0.19	0.00	$\dagger$: low success rates. $\ddagger$: infeasible branches.				
search	-	-	**0.83**	0.22	$\diamond$: imprecise dependency analysis.				

5.2 Efficiency

Table 3 contains the execution time required by `Austin` and `CAVM` to generate test data for studied functions. The dash (-) symbol denotes a complete timeout, that is, all 20 runs did not terminate within the timeout limit of 300 seconds (i.e., five minutes). In cases where neither tool reports a time out, the one with the shorter mean execution time is typeset in bold. Most of the factors that affected the branch coverage in Section 5.1 also affect the execution time:

– Indirect control dependency (□), low success rate (†), and infeasible branches (‡): repeated random restarts and timeouts caused by these add to the execution time, without contributing to the branch coverage.
– Large search spaces (∗): when the input variable range is reduced as in the Effectiveness study (Section 5.1), the average execution time of `CAVM` drops to about seven seconds, while retaining 100% branch coverage.
– Imprecise control dependency analysis (◊): with the explicit control dependency manually inserted, the average execution time of `CAVM` for the functions `delete`, `insert`, and `modify` in the `LinkedList` subject comes down to an average of 19 seconds. However, note that `Austin` can cover branches in `LinkedList` only after explicit `NULL` checks are added.

In comparison, a typical execution of `XYZXYZ` against all 22 functions from `decode.c` takes five seconds for static analysis and three seconds for pairwise test data generation. Similarly, functions in both `LinkedList` and `BinaryTree` take five and two seconds for each stage respectively.

Based on the results in Table 3, we can therefore answer RQ2: both `Austin` and `CAVM` take longer than `XYZXYZ`, whose execution time can be shorter by orders of magnitude. However, the speed of `XYZXYZ` comes at a trade-off with the coverage: while `CAVM` takes longer, it also covers more branches. Apart from its inability to cover `decode.c` functions, `Austin` can be slower than `XYZXYZ` despite using the same AVM technique.

6 Discussion: Insights and Experience Gained

We make the following observations based on our experience of the studied tools.

– Hybridisation can pay off: `XYZXYZ` is simple and fast, yet can be surprisingly effective for certain branches, including pointers. We expect many opportunities to hybridise this heuristic with more expensive techniques, such as AVM. The future integration of `CAVM` into `XYZXYZ` will follow a cascade model: `CAVM` will be applied only to the branches `XYZXYZ` misses.
– Direct source code instrumentation is not only possible but perhaps desirable: `CAVM` intentionally chose `clang` as the tool for instrumentation, because it is strong enough to be the frontend for a very widely used compiler. Compared to `CAVM`, the `CIL` transformation required by `Austin` often introduced build related issues, resulting in reduced usability.
– Pure search-based handling of dynamic data structures is possible: `CAVM` can instantiate dynamic data structures for test data generation. While the grow-and-search approach currently adopted by `CAVM` is not complete (e.g., backward pointers in doubly linked lists should not be *grown*), this paper shows that a pure search-based technique can be feasible.

7 Threats to Validity

Threats to internal validity includes the extent to which the results of the study supports the claims, such as the correctness of the tools used. `Clang` is the

frontend to one of the most widely used `C` compilers, LLVM-based `gcc` and withstood extensive public scrutiny. `XYZXYZ` is a commercial tool, and `Austin` is an open source tool based on peer-reviewed research outcome. `CAVM` itself will be publicly open sourced for further inspection.

Threats to external validity includes factors that may affect how well the conclusions generalise. Our conclusions are certainly limited by the choice of subjects and functions studied, and further generalisation is only possible based on wider empirical study. However, the current study provides sufficient evidence for the comparison of studied tools regarding specific and focused features of `C`, such as pointers and dynamic data structures.

8 Conclusion

We present `CAVM`, an AVM-based test data generation tool that handles dynamic data structures using a purely search-based approach. Unlike the current state of the art tool, `Austin`, which determines the shape of the required data structure using symbolic analysis, `CAVM` simply grows the data structure by successive pointer instantiations. The empirical comparison of `CAVM` against `Austin` and a commercial test data generation tool, `XYZXYZ`, shows that `CAVM` can cover many branches that neither of the other tools can. Future work include improvement of `CAVM` as well as its integration to `XYZXYZ`.

References

1. TIOBE Software: Tiobe Index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`
2. Barr, E.T., Harman, M., Jia, Y., Marginean, A., Petke, J.: Automated software transplantation. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. pp. 257–269. ISSTA 2015, ACM, New York, NY, USA (2015)
3. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. pp. 209–224. OSDI'08, USENIX Association, Berkeley, CA, USA (2008)
4. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Trans. Softw. Eng. 39(2), 276–291 (Feb 2013)
5. Fraser, G., Arcuri, A., McMinn, P.: A memetic algorithm for whole test suite generation. Journal of Systems and Software 103, 311 – 327 (2015)
6. Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated white-box test generation really help software testers? In: International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013. pp. 291–301 (2013)
7. Goues, C.L., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Proceedings of the 34th International Conference on Software Engineering. pp. 3–13 (2012)
8. Harman, M., Kim, S.G., Lakhotia, K., McMinn, P., Yoo, S.: Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In: Proceedings of the 3rd International Workshop on Search-Based Software Testing (SBST 2010). pp. 182 –191 (apr 2010)

9. Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Transactions on Software Engineering 30(1), 3–16 (Jan 2004)
10. Harman, M., Langdon, W.B., Jia, Y., White, D.R., Arcuri, A., Clark, J.A.: The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 1–14. ASE 2012 (2012)
11. Harman, M., McMinn, P.: A theoretical and empirical study of search based testing: Local, global and hybrid search. IEEE Transactions on Software Engineering 36(2), 226–247 (2010)
12. Harman, M., Wegener, J.: Evolutionary testing. In: Genetic and Evolutionary Computation (GECCO). Chicago (Jul 2003)
13. Horowitz, E., Sahni, S., Anderson-Freed, S.: Fundamentals of Data Structures in C. W. H. Freeman & Co., New York, NY, USA (1992)
14. Jin, W., Orso, A.: F3: Fault localization for field failures. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. pp. 213–223. ISSTA 2013, ACM, New York, NY, USA (2013)
15. Kempka, J., McMinn, P., Sudholt, D.: Design and analysis of different alternating variable searches for search-based software testing. Theoretical Computer Science 605, 1–20 (2015)
16. Korel, B.: Automated software test data generation. IEEE Transactions on Software Engineering 16(8), 870–879 (1990)
17. Lakhotia, K., Harman, M., Gross, H.: AUSTIN: A tool for search based software testing for the c language and its evaluation on deployed automotive systems. In: 2nd International Symposium on Search Based Software Engineering. pp. 101–110 (Sept 2010)
18. Lakhotia, K.: Open source code of Austin, `https://github.com/kiranlak/austin-sbst`, (Accessed 12/4/2017)
19. McMinn, P.: IGUANA: Input generation using automated novel algorithms. A plug and play research tool. Tech. Rep. CS-07-14, Department of Computer Science, University of Sheffield (2007)
20. McMinn, P., Kapfhammer, G.M.: AVMf: An open-source framework and implementation of the alternating variable method. In: International Symposium on Search-Based Software Engineering (SSBSE 2016). Lecture Notes in Computer Science, vol. 9962, pp. 259–266. Springer (2016), code and examples available at: http://avmframework.org
21. Miller, W., Spooner, D.L.: Automatic generation of floating-point test data. IEEE Transactions on Software Engineering 2(3), 223–226 (1976)
22. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs, pp. 213–228. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
23. Petke, J., Cohen, M.B., Harman, M., Yoo, S.: Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. IEEE Transactions on Software Engineering 41(9), 901–924 (September 2015)
24. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 263–272. ESEC/FSE-13, ACM (2005)

25. Tonella, P.: Evolutionary testing of classes. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 119–128. ISSTA '04, ACM, New York, NY, USA (2004)
26. Weimer, W., Nguyen, T., Goues, C.L., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE '09). pp. 364–374 (May 2009)
27. Yoo, S.: Evolving human competitive spectra-based fault localisation techniques. In: Fraser, G., Teixeira de Souza, J. (eds.) Search Based Software Engineering, Lecture Notes in Computer Science, vol. 7515, pp. 244–258. Springer Berlin Heidelberg (2012)