An Empirical Investigation Into Branch Coverage for C Programs Using CUTE and AUSTIN

Kiran Lakhotia[a], Phil McMinn[b], Mark Harman[a]

[a] *King's College London, CREST centre, Strand, London, WC2R 2LS, UK*
[b] *University of Sheffield, Regent Court, 211 Portobello, Sheffield, S1 4DP, UK*

Abstract

Automated test data generation has remained a topic of considerable interest for several decades because it lies at the heart of attempts to automate the process of Software Testing. This paper reports the results of an empirical study using the dynamic symbolic execution tool, CUTE, and a search based tool, AUSTIN on five non-trivial open source applications. The aim is to provide practitioners with an assessment of what can be achieved by existing techniques with little or no specialist knowledge and to provide researchers with baseline data against which to measure subsequent work. To achieve this, each tool is applied 'as is', with neither additional tuning nor supporting harnesses and with no adjustments applied to the subject programs under test. The mere fact that these tools can be applied 'out of the box' in this manner reflects the growing maturity of Automated Test Data Generation. However, as might be expected, the study reveals opportunities for improvement and suggests ways to hybridize these two approaches that have hitherto been developed entirely independently.

1. Introduction

Generating test inputs for software is a demanding problem. This paper focuses on test input generation to achieve branch coverage. Essentially, as with other test data generation problems, the underlying problem is one of reachability, which is of course undecidable, and so only partial coverage can be expected. Nonetheless, since the only available alternative is costly and unreliable human–based code coverage analysis, it makes sense to attempt to partly automate test input generation by seeking to develop algorithms and tools that can cover as much as possible in reasonable time. This has been the goal of Automated Test Data Generation for more than three decades, during which time great advances have been made in algorithms, techniques and tooling.

This paper concerns two leading approaches to software testing: dynamic symbolic execution and search based testing. Each has a long history of development dating back to the 1970s, but each has experienced a recent upsurge in interest, leading to the development of research tools.

Dynamic symbolic execution [1, 2, 3, 4, 5] is a development of symbolic execution based testing. The currently popular formulation of this approach originates in the seminal work of Godefroid *et al.* on Directed Random Testing [2].

Search Based Software Testing (SBST) [6] formulates the test data adequacy criteria as fitness functions, which can be optimized using Search Based Software Engineering [7, 8]. The search space is the space of possible inputs to the program under test. The objective function

captures the particular test adequacy criterion of interest. The approach is attractive because it is widely applicable; any testing problem for which a fitness function can be defined can be attacked with SBST. Of course, almost all approaches to testing involve some concept of test adequacy, which typically must be quantifiable in order to be applicable. As such, there is usually no shortage of candidate fitness functions [9].

This wide applicability has meant that a large number of applications of SBST have been developed. These include functional [10] and non-functional [11] testing, mutation testing [12], regression test selection [13], test case prioritization [14, 15], and interaction testing [16]. However, the most studied form of search based testing has been structural test data generation [17, 18, 19, 20, 21, 22, 23].

While many papers present results that demonstrate that both dynamic symbolic execution and search based techniques are better than random testing [1, 21, 23], there has been comparatively little work investigating and comparing their effectiveness with real world software applications. Previous work, has tended to be small-scale, considering only few programs, or considering only parts of individual applications to which the test generation technique is known in advance to be applicable. This is a reasonable consequence of the need to explore the technical development of new algorithmic features. For such developments, an empirical study will naturally focus on those aspects of programs under test that explore the performance and behaviour of the new technical contribution.

In previous studies, where the goal has been to explore technical developments, the test data generation tools are seldom, if ever, applied 'out of the box'; without customization or special handling for different test subjects or parameter tuning. This is not intended as a criticism of previous work. Since the topic of previous papers has tended to be the exploration of the behaviour of new technical contributions, it is only natural for the research questions to consider the performance of the new features under different conditions relating to customization and tuning.

Furthermore, no previous paper has attempted an extensive empirical report of comparative results from these two widely studied approaches to Automated Software Test Data Generation. Therefore, the current literature fails to provide convincing answers to several important questions that practicing software testers might well ask of the current state of the art. These include:

1. How effective are dynamic symbolic execution and search-based tools when applied with neither tuning nor customization to real-world software applications?
2. To what types of program are the dynamic symbolic execution and search-based approaches best suited?
3. How long will a tester have to wait for the results?

The aim of this paper is to provide answers to these questions. Fortunately, the state of the art in both Dynamic Symbolic Testing and Search Based Testing is now sufficiently mature that it is possible to do this. That is, we seek to take a step back from technical developments in order to consider what has been achieved overall; what is the current state of the art capable of automatically generating test inputs for structural testing?

Of course, the 'state of the art' is a continually developing body of knowledge and tools are continually under refinement. Since both fields of test data generation continue to attract a great deal of interest, new technical developments continue to emerge in the literature. Therefore, any attempt to answer these questions faces the problem of coping with a moving target. The results presented here are thus not intended to be definitive, once–and–for–all answers to these

questions. Rather, they are merely intended to capture a point in time in the development of this research agenda at which it was possible to provide realistic base line data.

The hope is that this base line data will be useful both for current practitioners and also for future research. It will allow the practitioner community to see what is possible and to which programs the techniques can be readily applied without specialist knowledge. It will support the research community in its ability to chart the development of the field.

As will be seen, the paper shows that despite considerable progress (which should make the current tooling available attractive to practitioners), there remain areas where test data generation can be improved (providing food for thought for the research community). The results also indicate that these two largely separate and independent approaches are to some degree complementary. This suggests the possible future development of hybrid approaches.

The paper reports the results of an empirical study which compares a dynamic symbolic execution tool, CUTE [1], and a search based tool, AUSTIN [24]. The test adequacy criterion under investigation is branch coverage. The primary contributions of this paper are the following:

1. An empirical study which determines the level of code coverage that can be obtained using CUTE and AUSTIN on the complete source code of five open source programs.
2. An empirical study investigating the wall clock time required for each tool to obtain the coverage that it does.
3. A more detailed technical assessment, based on the empirical study, of where CUTE and AUSTIN succeeded and failed.

We hope that the first of these contributions will provide useful base line data against which future developments in the field can be measured. The results indicate that there is plenty of room for such development, so there remain many interesting research challenges ahead. The aim of the final contribution is to provide an analysis of some of the challenges that remain for such improvement, distinguishing mere tool implementation issues from more fundamental algorithmic challenges.

The rest of the paper is organized as follows. Section 2 provides the background information to the dynamic symbolic execution and search based tools, CUTE and AUSTIN, which the empirical study presented in this paper uses. Section 3 outlines the motivation for our work, the research questions addressed, and the gap in the current literature this paper is trying to close. The empirical study, results and answers to the research questions are presented in Section 4, whilst Section 5 aims to clarify the difference between tool and technique related problems. Threats to validity are addressed in Section 6 and Section 7 presents related work. Section 8 concludes.

2. Background

Software testing can be divided into a sequence of three fundamental steps:

1. The design of test cases that are good at revealing faults, or which are at least adequate according to some test adequacy criterion.
2. The execution of these test cases.
3. The determination of whether the output produced is correct.

Sadly, in current testing practice, often the only fully automated aspect of this activity is test case execution. The problem of determining whether the output produced by the program under test is correct cannot be automated without an oracle, which is seldom available. Fortunately, the

problem of generating test data to achieve widely used notions of test adequacy *is* an inherently automatable activity. Such automation promises to have a significant impact on testing, because test data generation is such a time-consuming and laborious task. This paper focuses on structural test data generation.

Automated structural test data generation has been a burgeoning interest to researchers since at least the 1970s. In the 1970s two approaches to the problem emerged - symbolic execution [25], which is the basis of dynamic symbolic execution; and a method that reformulated the problem of executing a path through a program with floating-point inputs into objective functions [17], which later developed into the field known as search based testing. Both these techniques have witnessed a recent upsurge in interest as a result of the development of new approaches to symbolic execution, and a wider interest in search based optimization as a problem solving technique in Software Engineering. This section provides background information of each of the two techniques to make the paper self-contained.

2.1. Dynamic Symbolic Execution

Dynamic symbolic execution [1, 2, 3, 4] originates in the work of Godefroid *et al.* on Directed Random Testing [2]. It formulates the test data generation problem as one of finding a solution to a constraint satisfaction problem, the constraints of which are produced by a combination of dynamic and symbolic [25] execution of the program under test. Concrete execution typically drives the symbolic exploration of a program, and furthermore, dynamic variable values obtained by real program execution can be used to simplify path constraints produced by symbolic execution.

Symbolic execution involves constructing a system of constraints in terms of the input variables that describe when a program path will be executed. For example, the path condition which executes the `if` statement as true in Figure 1a would simply be $\langle a_0 + 5 = b_0 - 10 \rangle$, where a_0 and b_0 refer to the symbolic values of the input variables `a` and `b` respectively. The path condition is then solved by a constraint solver in order to derive concrete input values.

The path condition can easily become unsolvable, however, if it contains expressions that cannot be handled by constraint solvers. This is often the case with floating-point variables or non-linear constraints. For example, a linear constraint solver would encounter difficulties with the program of Figure 1b because of the non-linear predicate appearing in the `if` condition.

Dynamic symbolic execution may alleviate some of the problems associated with traditional symbolic execution by combining concrete execution with symbolic execution. The idea is to simplify the path condition by substituting sub-expressions with concrete values, obtained through actual dynamic executions. For example, this substitution process can be used to remove non-linear sub-expressions in a path condition, making them amenable to a constraint solver.

Due to the combination of concrete and symbolic execution, dynamic symbolic execution is also sometimes referred to as *concolic* testing. The term *concolic* was coined by Sen *et al.* [1] in their work introducing the CUTE tool. The remainder of this paper will use the terms *concolic testing* and *dynamic symbolic execution* interchangeably.

The CUTE Tool

CUTE implements a concolic testing strategy based on a depth-first exploration of all feasible program paths. The first path executed is that which is traversed with either all zero or, random inputs depending on CUTE's execution mode. The corresponding path condition forms the basis for successive iterations of the test data generation process.

Suppose the function in Figure 1b is executed with the random values `536` and `156` for `x` and `y` respectively. The path taking the false branch is executed and the corresponding path condition is $\langle x_0 * y_0 \geq 100 \rangle$, where x_0 and y_0 refer to the symbolic values of the input variables `x` and `y` respectively. The next execution path is chosen by inverting the last relational operator in the current path condition (*i.e.*, exploring the paths in a depth-first manner). Thus the new path condition becomes $\langle x_0 * y_0 < 100 \rangle$. Since this path condition contains non-linear constraints, CUTE replaces x_0 with its concrete value from the previous execution, *i.e.*, `536` [1]. The new, simplified path condition $\langle 536 * y_0 < 100 \rangle$is now linear and can be handled by CUTE's constraint solver (`lp_solve`) to find an appropriate value for `y` (*i.e.*, zero or any negative value). The exploration of program paths continues in this manner, inverting the last relational operator in a path condition (with backtracking), until all feasible execution paths through a unit have been explored. For functions with unbounded loops (or recursion), CUTE may unfold the body of the loop indefinitely. One can therefore place a limit on the length of a path condition by bounding CUTE's depth first search via a command line parameter.

2.2. Search Based Testing

Like symbolic-execution-based testing, the first suggestion of optimization as a test data generation technique also emerged in the 1970s, with the seminal work of Miller and Spooner [17]. Miller and Spooner showed that the series of conditions that must be satisfied for a path to be executed can be reformulated as an *objective function*, the optima of which (*i.e.*, the test data that executes the path) could be found using optimization techniques.

The role of an objective function is to return a value that indicates how 'good' a point in a search space (*i.e.*, an input vector) is compared to the best point (*i.e.*, the required test data); the global optimum. For example, if a program condition `a == b` must be executed as true, the objective function could be $|a - b|$. The closer the output of this formula is to zero, the 'closer' the program input is to making `a` and `b` equal, and the closer the search technique is to finding the test data of interest.

Because an optimizing search technique is used rather than a constraint solver, non-linear constraints present fewer problems. For example the surface of the objective function for taking the true path through the `if` condition of Figure 1b can be seen in Figure 1c. The surface is smooth and provides the optimization process with a clear 'direction', guiding the search to the required test data. Furthermore, computation of the objective function by dynamically executing the program alleviates another problem of both symbolic and concolic testing, *i.e* floating-point inputs.

The suggestions of Miller and Spooner were not subsequently taken up until Korel developed them further in 1990 [18], when he proposed the use of a search technique known as the 'alternating variable method'. Since then the ideas have been applied to other forms of testing [10, 11, 12, 13, 14, 15, 16], using a variety of optimizing search techniques, including genetic algorithms [19, 20, 21]. The objective function has been further developed to generate test data for a variety of program structures, including branches, as well as paths [21].

The AUSTIN Tool

AUSTIN is a tool for generating branch adequate test data for C programs. It is an improved version of our earlier work [24], most notably in the way pointer constraints are solved. Our earlier work did not include feasibility checks of pointer constraints because it did not use an equivalence graph to solve them.

AUSTIN does not attempt to execute specific paths in order to cover a target branch; the path taken up to a branch is an emergent property of the search process. The objective function used by AUSTIN was introduced by Wegener *et al.* [21] for the Daimler Evolutionary Testing System. It evaluates an input against a target branch using two metrics; the *approach level* and the *branch distance*. The approach level records how many nodes on which the branch is control dependent were not executed by a particular input. The fewer control dependent nodes executed, the 'further away' the input is from executing the branch in control flow terms. Thus, for executing the true branch of statement 3 in Figure 1d; the approach level is

- 2 when an input executes the false branch of statement 1;
- 1, when the true branch of statement 1 is executed followed by the false branch of statement 2;
- zero if statement 3 is reached.

The branch distance is computed using the condition of the decision statement at which the flow of control diverted away from the current 'target' branch. Taking the true branch from statement 3 as an example again, if the false branch is taken at statement 1, the branch distance is computed using $|a - b|$, whilst $|b - c|$ is optimized if statement 2 is reached but executed as false, and so on. The branch distance is normalized (see formula in Figure 2) and added to the approach level. The search method used is the Alternating Variable Method (AVM), proposed by Korel [18]. The AVM is a simple search technique, which was shown to be very effective by Harman and McMinn [23] when compared with more sophisticated optimization techniques such as genetic algorithms.

AUSTIN, like CUTE, begins with all primitives set to zero. If the target is not executed, the AVM cycles through each input of primitive type and performs so-called 'pattern moves', guided by the objective function. If a complete cycle of adjustments takes place with no improvement in objective value, the search restarts using random values.

Suppose the program of Figure 1d is executed with the input $\langle a = 100, b = 200, c = 300 \rangle$, with the aim of executing the true branch of statement 3. The AVM takes the first variable, `a`, and performs *exploratory moves*; executions of the program where `a` is decreased and increased by a small amount δ ($\delta = 1$ for integers and 0.1 for floating point variables). An increased value of the variable `a` brings it closer to `b` and results in a better objective value.

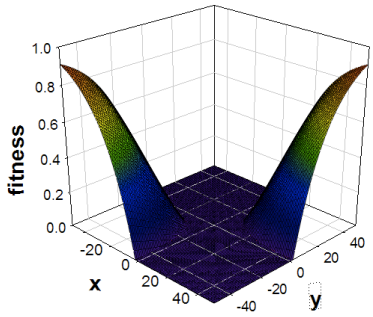
The AVM then makes *pattern moves* for as long as the objective function continues to yield an improved value. The value added to the variable in the nth pattern move is computed using the formula $2^n \cdot dir \cdot \delta$; where $dir \in \{-1, 1\}$ corresponding to the positive or negative 'direction' of improvement, identified by the initial exploratory moves. Thus consecutive exploratory moves for the variable `a` are 102, 104, 108 and so on. Pattern moves will improve the objective value until a value of 228 is reached for the variable `a`. At this point the minimum ($a = 200$) has been overshot, so the AVM repeats the exploratory-pattern move cycle for as long as necessary until the minimum is reached. When $a = 200$, the true branch of statement 1 is executed. For executing statement 2, exploratory moves on the variable `a` both lead to a worse objective value, because the original outcome at statement 1 is affected and the approach level worsens. The AVM will then consider the next variable, `b`. Exploratory moves here have the same effect, and so the AVM moves onto the variable `c`. Decreasing the value of `c` improves the objective value, and so pattern moves are made. Eventually each input value is optimized to 200.

```
void testme1(int a, int b)
{
    a += 5; b -= 10;
    if (a == b)
        // ...
}
```

(a) *Example for demonstrating symbolic execution. The branch predicate is linear*

```
void testme2(int x, int y)
{
    if (x * y < 100)
        // ...
}
```

(b) *Example for demonstrating concolic execution and search based testing. The branch predicate is non-linear*



(c) *Surface of the objective function for executing the true branch of the program in (b)*

```
    void testme3(int a, int b, int c)
    {
(1)     if (a == b)
(2)         if (b == c)
(3)             if (a == c)
(4)                 // ...
    }
```

(d) *Example for demonstrating objective function calculation for the AUSTIN tool*

Figure 1: Examples for demonstrating symbolic, concolic and search based testing.

$$f(dist) = 1 - 1.001^{-dist}$$

Figure 2: Normalization function used in AUSTIN for branch distances. *dist* is the branch distance value described in Section 2.2.

Should exploratory moves produce no improvement in objective value, with the required test data not found either, the search has hit a local minima from which it cannot proceed. AVM terminates and restarts with a new random input vector. Typically, the search is afforded a 'budget' of objective function evaluations (*i.e.*, program executions) in which to find test data, otherwise the search is deemed to have failed. This could be because the branch is infeasible. In some cases, however, the objective function surface can be flat, offering the search no guidance to the required test data. This is sometimes due to the presence of boolean 'flag' variables [26, 27], which can result in two plateaus for a branch condition; one where the flag is true, and one where it is false.

2.3. Handling Inputs Involving Pointers

The above descriptions explained how concolic testing and search based testing handle inputs of a primitive type only. This section explains how CUTE and AUSTIN handle pointer inputs. The CIL (C Intermediate Language) infrastructure [28] is used to transform and simplify programs such that all predicates appearing in decision statements contain only either pointer types or arithmetic types. In addition, the source contains no compound predicates. Thus all conditions involving pointers are of the form `x == y` or `x != y` (where `x` and `y` could also represent $NULL$).

Pointers in CUTE

The first path explored by the CUTE tool is the path executed where inputs of primitive type are zero (or of a random value, depending on the settings used). If the function involves pointer variables, these are always initially set to $NULL$. However, further paths through the program may require pointers to point to a non-null data structure. In order to find the 'shape' of this data structure, CUTE incorporates symbolic variables for pointers in the path condition. A graph-based process is used to check that the constraints over the pointer variables are feasible, and finally, a simple procedure is used to build the data structure required.

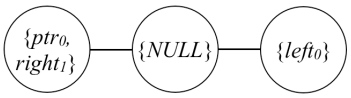
For the program of Figure 3a, and the path that executes the true branch at each decision, CUTE accumulates the path constraint:

$$ptr_0 \neq NULL \wedge left_0 \neq NULL \wedge right_1 = ptr_0$$

CUTE keeps a map of which symbolic variable corresponds to which point in the data structure, for example, $left_0$ maps to `ptr->left`. The feasibility check involves the construction of an undirected graph, which is built incrementally at the same time as the path condition is constructed from the conditions appearing in the program. The nodes of the graph represent abstract pointer locations, with node labels representing the set of pointers which point to those locations. A special node is initially created to represent $NULL$. Edges between nodes represent inequalities. After statement 1 in the example, the graph consists of a node for ptr_0 with an edge leading from it to the $NULL$ node. When statement 2 is encountered, a new node is constructed for $left_0$, with an edge to $NULL$. Finally, $right_1$ is merged into the existing ptr_0 node, as they must point to the same location (Figure 3b). Feasibility is checked as each constraint is added for each decision statement. An equality constraint between two pointers `x` and `y` is feasible if and only if there is no edge in the graph between nodes representing the locations of `x` and `y`. An inequality constraint between `x` and `y` is feasible if and only if the locations of `x` and `y` are not represented by the same node.

```
        void testme4(item* ptr)
        {
(1)         if (ptr != NULL)
(2)             if (ptr->left != NULL)
(3)                 if (ptr->left->right == ptr)
(4)                     // ...
        }
```

(a) *Code snippet*



(b) *CUTE feasibility graph for path which executes all decisions in (a) as true*

Figure 3: Example for demonstrating pointer handling in CUTE and AUSTIN.

Table 1: Dynamic data structure creation according to individual constraints encountered along the path condition for CUTE and AUSTIN.

Constraint	**CUTE**	**AUSTIN**
$m_0 = NULL$	Assign $NULL$ to m_0	
$m_0 \neq NULL$	Allocate a new memory location pointed to by m_0	
$m_0 = m_1$	Make m_1 alias m_0	
$m_0 \neq m_1$	Allocate a new memory location pointed to by m_1	With an even probability, assign $NULL$ or allocate a new memory location pointed to by m_0

If the path condition is feasible, the data structure is built incrementally. Each new branching decision adds a new constraint to the path condition, and the data structure is created on the basis of each constraint using the rules of Table 1. A more detailed treatment can be found in the work of Sen *et al.* [1].

Pointers in AUSTIN

There has been little work with respect to generating dynamic data structures in search based testing. Korel [18] developed a limited method for simple Pascal data structures. In order to apply search based testing to real world programs this limitation had to be overcome. AUSTIN uses search based techniques for primitive inputs, a symbolic process akin to that of CUTE is used for pointers [24]. As with CUTE, pointer inputs are initially set to $NULL$. During the search process, a branch distance calculation may be required for a condition that involves a pointer comparison. However branch distances over physical pointer addresses do not usually give rise to useful information for test data generation; for example it is difficult to infer the shape of a dynamic data structure. Therefore, instead of computing a branch distance, AUSTIN

Table 2: Details of the test subjects used in the empirical study. In the 'Functions' column, 'Non-trivial' refers to functions that contain branching statements. 'Top-level' is the number of non-trivial functions that a test driver was written for, whilst 'tested' is the number of functions that were testable by the tools (*i.e.*, top-level functions and those that could be reached interprocedurally). In the 'Branches' column, 'tested' is the number of branches contained within the tested functions.

Test	**Lines**	**Functions**				**Branches**	
Object	**of Code**	Total	Non-Trivial	Top Level	Tested	Total	Tested
`libogg`	2,552	68	33	32	33	290	284
`plot2d`	6,062	35	35	35	35	1,524	1,522
`time`	5,503	12	10	8	10	202	198
`vi`	81,572	474	399	383	405	8,950	8,372
`zile`	73,472	446	339	312	340	3,630	3,348
Total	169,161	1,035	816	770	823	14,596	14,084

performs symbolic evaluation for the path taken up to the predicate as executed by the current input generated by the AVM. The result is a path condition of the same form as generated by CUTE. As with CUTE, the constraints added to the path condition are used to incrementally build the dynamic data structure. The rules for doing this appear in Table 1. AUSTIN does not perform a feasibility check; if the branching condition at the control dependent node is not executed as required, the AVM process restarts afresh. For a more in-depth treatment, see reference [24].

3. Motivation and Research Questions

One of the first tests for any automatic test data generation technique is that it outperforms random testing. Many authors have demonstrated that both concolic based and search based techniques can outperform purely random test data generation. However, there are fewer studies that have attempted to evaluate concolic and search based approaches on real world programs.

Previous studies have tended to be small-scale [18, 21] or, at least in the case of search based approaches, concentrated on small 'laboratory' programs. Where production code has been considered, work has concentrated solely on libraries [22] or individual units of applications [23]; usually with the intention of demonstrating improvements or differences between variants of the techniques themselves.

Studies involving concolic approaches have also tended to focus on illustrative examples [1, 29, 30, 3], with relatively little work considering large scale real world programs such as the `vim` text editor [5, 31], network protocols [32] or windows applications [33]. Furthermore, no studies have compared the performance of concolic and search based testing on real world applications.

The research questions to be answered by the empirical study are therefore as follows:

RQ 1: How effective are CUTE and AUSTIN for real world programs? Given a set of real world programs, how good are CUTE and AUSTIN at achieving structural coverage?

RQ 2: What is the relative efficiency of each tool? If it turns out that both tools are similarly effective at generating test data, efficiency will be the next issue of importance as far as a practitioner is concerned. If one tool is more effective but less efficient than the other, what is the trade off that the practitioner has to consider?

RQ 3: Which types of program structure did each tool fail to cover? Which functions could the tools not handle, and for the functions which could be handled, what types of branches remained stubborn to the test generation process? What are the challenges that remain for automatic test data generation tools?

4. Empirical Study

The empirical study was performed on a total of 169,161 pre-processed lines of C code contained within five open-source programs. This is the largest study of search based testing by an order of magnitude and is similar in size to the largest previous study of any form of concolic testing.

4.1. Test subjects

Details of the subjects of the empirical study are recorded in Table 2. A total of 770 functions were tested. Since the study is concerned with branch coverage, trivial functions not containing any branches were ignored. In addition, further functions had to be omitted from the study because they could not be handled by CUTE or AUSTIN. These included functions whose inputs were files, data structures involving function or `void` pointers, or had variable argument lists. These problems are discussed further in Section 4.3 in the answer to research question 3.

The programs chosen are not trivial for automated test data generation. `libogg` is a library used by various multimedia tools and contains functions to convert to and from the *Ogg* multimedia container format, taking a bitstream as input. `plot2d` is a relatively small program which produces scatter plots directly to a compressed image file. The core of the program is written in ANSI C, however the entire application includes C++ code. Only the C part of the program was considered during testing because the tools handle only C. `time` is a GNU command line utility which takes, as input, another process (a program) with its corresponding arguments and returns information about the resources used by a specific program, including wall-clock and CPU times. `vim` is a common text editor in Unix, and makes heavy use of string operations; as does `zile`, another GNU program, designed to be a more lightweight text editor than Emacs.

4.2. Experimental Setup

Each function of each test subject was taken in turn (hereinafter referred to as the 'FUT' - Function Under Test), with the aim of recording the level of coverage that could be achieved by each tool.

Since CUTE and AUSTIN take different approaches to test data generation, care had to be taken in setting up the experiments such that the results were not inadvertently biased in favour of one of the tools. The main challenge was identifying suitable stopping criteria that were 'fair' to both tools. Both tools place limits on the number of times the function under test can be called, yet this is set on a per-function basis for CUTE and a per-branch basis for AUSTIN. Furthermore, one would expect CUTE to call the function under test less often than AUSTIN, because it carries out symbolic evaluation. Thus, setting a limit that was 'equal' for both tools was not feasible. Therefore it was decided that each limit would be set to a high value, with a time limit of 2 minutes of wall clock time per FUT used as an additional means of deciding when a tool's test data generation process should be terminated.

CUTE's limit was set to the number of branches in the FUT multiplied by 10,000. CUTE can reach this limit in only two cases; firstly if it keeps unfolding a loop structure, in which case it

won't cover any new branches; or secondly if the limit is less than the number of interprocedural branches, *i.e.*, the total number of branches reachable from the FUT (which was not the case for any of the test subjects considered). AUSTIN's limit was set to 10,000 FUT executions per branch, with branches in the FUT attempted sequentially in reverse order. Oftentimes the search process did not exhaust this limit but was terminated by the overall time limit instead.

Thirty 'trials' were performed for each tool and each function of each test subject. AUSTIN is stochastic in nature, using random points to restart the search strategy once the initial search, starting with all primitives as zero, fails. Thus, several runs need to be performed to sample its behaviour. Since some test subjects exhibit subtle variations in behaviour over different trials (*e.g.* in the `time` program), CUTE was also executed thirty times for each function, so that AUSTIN did not benefit unfairly from multiple executions. For example, some branches of the `time` function (transitively) depend on the result of computations involving the `gettimeofday` function.

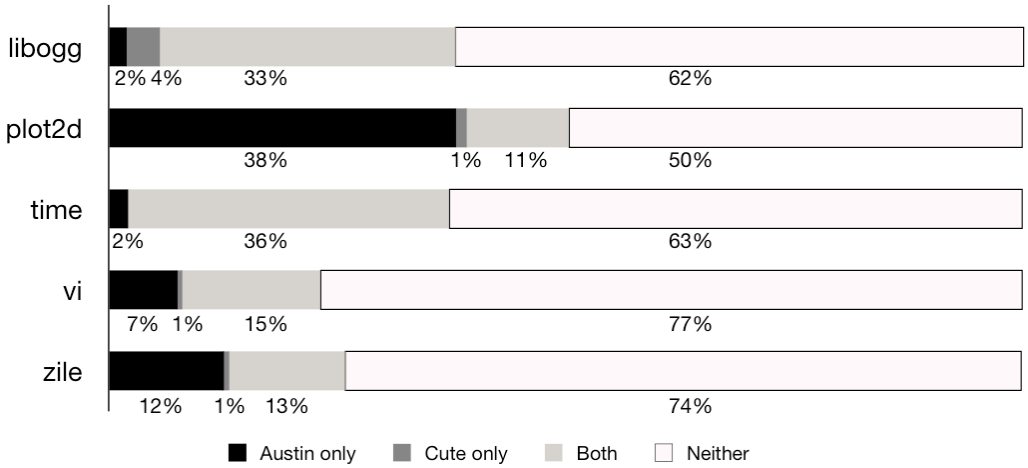
Coverage was measured in two ways. The first is respective to the branches covered in the FUT only. A branch is counted as covered if it is part of the FUT, and is executed at least once during the thirty trials. The second measure takes an interprocedural view. A branch is counted as covered if it is part of the FUT or any function reachable through the FUT. Interprocedural coverage is important for CUTE, since path conditions are computed in an interprocedural fashion. Any branches covered interprocedurally by AUSTIN, however, are done so serendipitously, as the tool only explicitly targets branches in the FUT.

Apart from the settings necessary for a fair comparison, as discussed above, both tools were applied 'out of the box', *i.e.*, with default parameters and without the writing of special test drivers for any of the test subjects. As mentioned in Section 2.1, CUTE has an option to limit the level of its depth-first search, thus preventing an infinite unfolding of certain loops. However, as it is not generally known, *a priori*, what a reasonable restriction is, CUTE was used in its default mode with no specified limit, *i.e.*, an unbounded search.

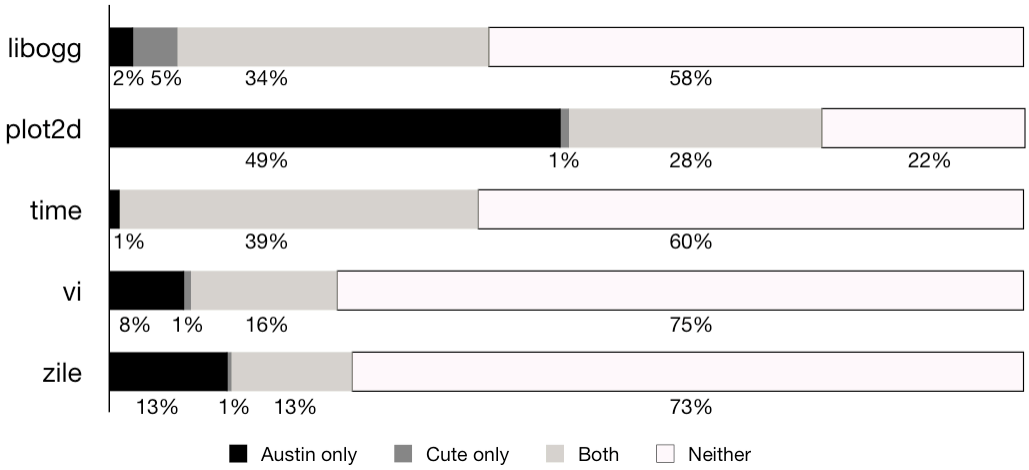
The test driver for both tools is not only responsible for initializing input parameters but also the place to specify any pre-conditions for the function under test. AUSTIN generates a test driver automatically by examining the signature of the FUT. The test drivers for CUTE had to be written manually but were constructed using the same algorithm as AUSTIN. Writing pre-conditions for functions without access to any specifications is non-trivial. For the study only the source code was available with no other documentation (including virtually no comments in the source code). Thus, pre-conditions of functions would have had to be inferred manually by inspection, clearly not a feasible task. Yet, without capturing a functions' pre-condition in a test driver, a tool is likely to produce 'invalid' inputs, especially for pointer type inputs. Preliminary experiments showed that a common pre-condition of functions was that pointers do not point to *NULL* in order to prevent *NULL*-pointer dereferences. Therefore it was decided the only pre-condition to use was to require top level pointers to be non-*NULL* (as described in Section 2.3).

4.3. Answers to Research Questions

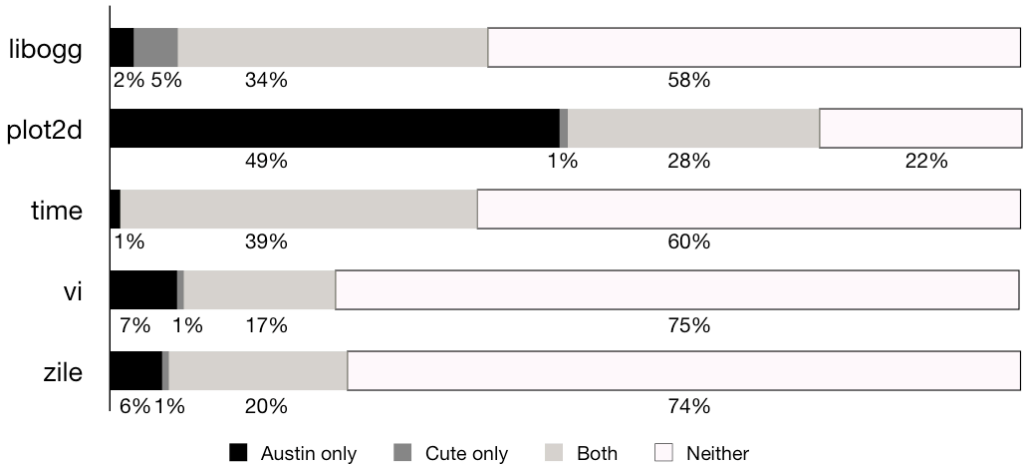
RQ 1: How effective are CUTE and AUSTIN for real world programs? Figure 4 plots three different 'views' of the coverage levels obtained by CUTE and AUSTIN with the test subjects. The first view, Figure 4a, presents coverage of branches in the FUT only. However, CUTE places an emphasis on branches covered in functions called by the FUT, building up path conditions interprocedurally. For AUSTIN interprocedural branch coverage is incidental, with test generation directed at the FUT only. Therefore, Figure 4b plots interprocedural coverage data which, in theory, should be favourable to CUTE. Finally, CUTE could not attempt 138 functions, containing



(a) *Branches covered only as part of the function under test*



(b) *Branches covered in the function under test and interprocedurally*



(c) *Branches covered in functions that CUTE can handle only*

Figure 4: Branch coverage for the test subjects with CUTE and AUSTIN. Graph (a) counts only branches covered in each function tested individually. Graph (b) counts branches covered in the function under test and branches covered in any functions called. Graph (c) is graph (b) but with certain functions that CUTE cannot handle excluded.

Table 3: Comparing wall clock time and interprocedural branch coverage for a sample of branches. FUT refers to the Function Under Test and IP to the number of Interprocedural branches, *i.e.*, the number of branches reachable from the FUT (including the branches in the FUT itself).

FUT	**Branches**		**CUTE**			**AUSTIN**		
			Time	**Covered**		**Time**	**Covered**	
	FUT	IP	**(s)**	FUT	IP	**(s)**	FUT	IP
libogg:								
`ogg_stream_clear`	8	10	0.84	7	10	134.75	5	7
`oggpack_read`	14	14	0.24	2	2	0.18	2	2
plot2d:								
`CPLOT_BYTE_MTX_Fill`	4	8	131.25	4	4	130.05	1	1
`CPLOT_DrawDashedLine`	56	56	130.43	13	13	130.40	37	37
`CPLOT_DrawPoint`	16	16	0.51	13	13	131.82	13	13
time:								
`resuse_end`	6	6	0.42	4	4	131.84	5	5
vi:								
`_compile`	348	348	2.11	26	26	34.14	24	24
`exitex`	4	4	146.47	1	1	0.22	1	1
`plod`	174	174	1.58	18	18	137.17	18	18
`pofix`	4	4	0.41	1	1	135.08	1	1
`vappend`	134	426	0.53	6	6	0.26	6	6
`vgetline`	220	228	0.81	3	3	0.26	3	3
`vmain`	404	480	0.27	3	3	0.3	3	3
`vmove`	30	30	0.79	3	3	0.24	3	3
`vnpins`	6	108	0.22	2	2	0.25	2	2
`vputchar`	148	212	0.2	4	4	0.25	4	4
zile:								
`astr_rfind_cstr`	6	6	0.45	2	2	0.17	2	2
`check_case`	6	6	0.38	1	1	130.49	6	6
`expand_path`	82	84	0.37	0	1	0.16	0	1
`find_window`	20	20	0.4	1	1	131.4	1	1
Total	1,690	2,240						

1,740 branches. Some of these functions involved `void` pointers, which cannot be handled by CUTE. However, a number of functions could not be tested by CUTE because the test subject did not compile after CUTE's instrumentation. For certain functions of the `zile` test subject, the instrumentation casts a data structure to an unsigned integer, and subsequently tries to dereference a member of the data structure, which results in an error. Since CUTE's exploration behaviour is interprocedural, all functions within this source file became untestable. Thus, Figure 4c plots interprocedural branch coverage, but removing these branches from consideration.

Strikingly, all three views of the coverage data show that in most cases, the majority of branches for an application were not covered by *either* tool. The only exception is the `plot2d` test subject. Here, AUSTIN managed 77% coverage taking interprocedural branches into account, compared to CUTE's 29%. Code inspection revealed that 13 functions in `plot2d` contained unbounded loops. Whenever a loop exit condition of an unbounded loop forms the last (yet) un-inverted condition, CUTE will keep inverting the same condition over and over again. It therefore never attempted to cover any more branches in the unit and instead kept increasing the

```
void foo(int n, ...) {
   for(int i = 0; i < n; i++) ...
}
```

Figure 5: Example used to illustrate how unbounded loops can slow down the test data generation process, thus leading to a premature timeout after 2 minutes.

number of loop iterations by one until its timeout or iteration limit was reached. For all other subjects, coverage for either tool does not exceed 50% whatever 'view' of the data is considered. It has to be noted, however, that AUSTIN does seem to cover a higher number of the branches. When a modified path condition falls outside the supported theory of CUTE's constraint solver, unlike AUSTIN, CUTE does not try a fixed number of random 'guesses' in order to find test data. AUSTIN on the other hand will spend 10,000 attempts at covering the branch. In the worst case this is equal to performing 10,000 random searches. Nevertheless, this gives AUSTIN a higher chance of finding test data than CUTE.

Section 5 will examine in more detail if the observed results are a consequence of the techniques implemented by AUSTIN and CUTE or due to limitations of the tools. First though, the efficiency of each of the tools is examined with respect to a subset of the branches before research question 3 aims to identify where the problems were and where the challenges remain.

RQ 2: What is the relative efficiency of both tools? In order to answer this research question, a random sample of 20 functions (from the programs in Table 2) were taken and the performance of each individual tool analysed further. These functions are listed in Table 3 and comprise 1690 branches, with a further 550 reachable interprocedurally.

CUTE times out (reaching the 120 second limit) on three occasions. This is because CUTE gets stuck unfolding loops in called functions. AUSTIN times out on eight occasions. For example, the function `ogg_stream_clear` from `libogg` takes as input a pointer to a data structure containing 18 members, one of which is an array of 282 unsigned characters, while 3 more are pointers to primitive types. Since AUSTIN does not use any input domain reduction, it has to continuously cycle through a large input vector in order to establish a direction for the search so it can start applying its pattern moves. The second cause of timeouts was indirectly related to unbounded loops. If the number of loop iterations is controlled by an input parameter to the function under test, AUSTIN will, on average, assign a large number to this parameter. Consider the example in Figure 5.

After a random restart, AUSTIN will pick any value for `n` from the range of [INT_MIN,INT_MAX]. Assigning large values to `n` can slow down the overall execution of the function under test and thus increase the wall clock time of the test data generation process.

Overall the table indicates that CUTE is more efficient than AUSTIN, though the results very much depend on the function under test as well as the execution mode of CUTE. Using CUTE with an unbounded depth-first search on units containing unbounded loops (or recursion) is very inefficient. For all other functions CUTE is very efficient, in large parts due to its relatively simple constraint solver (`lp_solve`) and its own internal optimization steps. These ensure calls to the constraint solver are minimized whenever possible.

AUSTIN is a prototype research tool and thus not optimized for performance. Despite this, AUSTIN terminates within a second for many functions. For the functions where AUSTIN timed out, its efficiency could have been improved by incorporating an input domain reduction

```
void foo(int a) {
   if( a == 4 )
     //target
}
```

Figure 6: Example used to illustrate the relative inefficiency of AUSTIN for certain branches.

as proposed by Harman *et al.* [34].

We speculate however that in general a constraint solver will be faster than the search method implemented in AUSTIN (though perhaps at the cost of effectiveness). Consider the example from Figure 6. A concolic testing tool will be very fast in achieving 100% branch coverage of the function in Figure 6, having to execute the function twice to achieve its coverage. AUSTIN on the other hand requires 7 executions of the function in order to achieve 100% branch coverage (assuming AUSTIN starts with primitives initialized to zero).

RQ 3: Which types of program structure did each tool fail to cover?

Table 4: Errors encountered during test data generation for the sample of branches listed in Table 3.

	CUTE	**AUSTIN**
Timeout	91	277
Aborted	0	113
Segmentation fault	419	180
Floating point error	60	90
Failed to take predicted path	30	*n/a*

Table 4 logs common reasons why the tools terminated abnormally for the 23 sample functions shown in Table 3.

Segmentation faults. CUTE terminated prematurely 419 times, and AUSTIN 180 times because of segmentation faults. These faults are the result of implicit constraints on pointers (*i.e.*, missing guarding statements) and a consequence of assigning 'bad' values to input parameters. Consider the example in Figure 7 (taken from `libogg`). The variable `vals` is an indication of the number of elements in the array `b`. If this relationship is not observed during testing, *e.g.* by assigning $NULL$ to `b` and setting `vals` > 0, the program will crash with a segmentation fault.

Floating point exceptions. Another cause of system crashes for both tools were floating point exceptions. The IEEE 754 standard classifies 5 different types of exceptions: invalid operation; division by zero; overflow; underflow; inexact calculation.

During the sample study CUTE raised 60 floating point exceptions and AUSTIN 90. The signal received for both tools was $SIGFPE$, which terminates the running process. Code inspection revealed that in all cases the cause of the exception was a division by zero error. This is not surprising since all primitive inputs are initialized to zero by both tools.

Unhandleable functions. Out of all functions considered during the main study, 46 could not be tested by both tools and an additional 138 not by CUTE because of unhandled input types. AUSTIN, for example, initializes void pointers to $NULL$ and does not attempt to assign any other value to them. CUTE cannot handle such pointers at all, and generates an undefined

```
void
cliptest(unsigned long *b,int vals,...) {
  ...
  for(i=0;i<vals;i++)
    oggpack_write(&o,b[i],bits?bits:ilog(b[i]));
  ...
}
```

Figure 7: CUTE and AUSTIN require a human to write preconditions to functions. This can be a laborious task, especially in the context of function calls made from within the function under test. This examples illustrates the need for tools to be able to infer simple preconditions, such as *NULL* pointer checks, in order to avoid program crashes, from which neither CUTE nor AUSTIN can recover.

`_cu__unhandledInputType` call. It is not clear what would be a good strategy for setting such pointers. Source code analysis may be required to determine the possible data types.

A similar problem to void pointers can be observed with the `va_list` type, allowing variable length argument functions in C. This type is essentially a linked list container and the data types of the data items cannot always be established a priori. An implementation issue is also related to variable argument lists. A call to `__builtin_va_arg` in the version of the `glibc` library, takes the data type:

```
__builtin_va_arg(marker, mytype)
```

as its second parameter. CIL transforms such calls into an internal representation of

```
__builtin_va_arg(marker, sizeof(mytype), &x);
```

where `x` is a variable of `mytype`. When printing the source code from CIL, the printer will try to print the original code. Due to the code simplification transformations used, the internal CIL form is printed instead, causing a compilation error.

Out of all available functions in the preprocessed C code, 19 had a function pointer as one of their formal parameters. This is another input type that causes problems for both CUTE and AUSTIN. Currently the tools lack the infrastructure to either select a matching function from the source code, or, to generate a stub which matches the function signature. One way to address the problem would be to manually supply the address of a function, but then the test data generation process would not be fully automated.

A special case exists for AUSTIN when a function pointer is declared as a member of an input data structure. AUSTIN treats such members like any other pointer and will assign *NULL* or a block of newly allocated memory to the pointer during input initialization. Of course this will crash the program as soon as the application attempts to use the function pointer.

Another implementation related drawback for both CUTE and AUSTIN [1] is the failure to handle bitfields in input data structures. Both tools use the address of variables during input initialization (see the example in Figure 8), and thus attempt to take the address of a bitfield when generating inputs. This is, of course, not possible and results in the `gcc` compilation error "cannot take address of bit-field 'bf' ".

[1] This has been fixed in the current version of AUSTIN so that it now supports bitfields.

```
typedef struct mystruct {unsigned int bf : 5;}
mystruct;

//CUTE / AUSTIN:
_IUInt(
 (unsigned long)&(((struct mystruct*)maddress)->bf));
```

Figure 8: Example used to illustrate that CUTE and AUSTIN cannot handle code which contains bitfields.

Many real world programs contain some form of file I/O operation. Yet neither tool fully supports functions whose parameter list contains a pointer to a file structure. During this study, the `FILE` data structure contained incomplete type information in the accessible source code, and thus neither tool had the information required to instantiate such inputs. Even if all the type information had been available, the tools would not have been able to generate 'valid' inputs. This is because each member of the data structure has a semantic meaning. Therefore it would not make sense to populate it with random values. The same applies to any `_IO_FILE` data structure (*e.g.* as used by `stderr` or `stdout`). Populating these with arbitrary values will likely result in nothing more than program crashes, or extensive testing of error handling procedures.

CUTE-specific issues. During the study, CUTE failed to cover branches for the following reasons. As explained in our experimental setup, only the source file containing the current function under test was instrumented. Any other source files of the program remained untouched, and thus formed a 'black box' library as far as CUTE was concerned. Black box functions (system or application specific) influence CUTE's effectiveness in two ways.

Return values from black box functions cannot be represented symbolically and are thus treated as constants in a path condition where applicable. The underlying assumption is that their value remains unchanged over successive iterations with similar inputs. If this is not the case, successive executions may not follow the sub-path predicted by CUTE (which results in CUTE aborting its current search, reporting "... CUTE failed to take the predicted path", and restarting afresh).

Worse still, formal parameters of the function under test may be passed by reference to a black box function which, in turn, modifies their value. Since CUTE cannot be aware of how these inputs may change, it is likely that solutions generated by CUTE do not satisfy the real path conditions and thus also result in prediction errors by CUTE.

An illustrative example can be found in the `time` program, involving the function `gettimeofday`, which populates a data structure supplied as input parameter to the function. While CUTE attempts to set members of this structure in order to evaluate the predicate shown in Figure 9 with a true outcome, any values of the data structure are overridden before reaching the predicate. A weakness of CUTE is that it fails to recognize the indirect assignment to the input data structure via the function `gettimeofday`. A better approach would be to assume any pointer passed to an uninstrumented piece of code may be used to update blocks of memory reachable through that pointer.

One of the claimed strengths of concolic testing is its ability to overcome weaknesses in constraint solvers by simplifying symbolic expressions so they can be handled through existing theories. The light-weight solver used by CUTE requires many such simplifications. CUTE only handles additions, subtractions, (side effect free) assignments and linear multiplication state-

```
resuse_end (pid_t pid, RESUSE *resp) {
 ...
 gettimeofday (&resp->elapsed,(struct timezone *) 0);
 ...
 if (resp->elapsed.tv_usec < resp->start.tv_usec)
  //target
}
```

Figure 9: Example used to illustrate how black box function calls can inhibit dynamic symbolic execution based techniques.

```
BOOL CPLOT_DrawDashedLine(...,const int x, ...
  const int to_x,
  const int to_y, ...) {
  ...
  if( to_x - x == 0 ) {
    for( row=y; row<=to_y; row++ ){ ... }
    return TRUE;
  }
  //CODE NEVER EXECUTED BY CUTE
}
```

Figure 10: Example used to illustrate how CUTE can get stuck unfolding loops when used with an unbounded depth-first exploration strategy.

ments symbolically. Non-linear expressions are approximated as described in Section 2.1, while all other statements (*e.g.* divisions) are simply ignored (in symbolic terms). Inspection of the source code for the different programs revealed that path conditions become very quickly dominated by 'constant' values due to the nature of operations used in the code. The lack of symbolic information means inverted constraints quickly become infeasible within the given path condition. Thus CUTE is restricted in the number of execution paths it can explore.

Finally CUTE failed to cover certain branches because of an unbounded DFS (CUTE's default mode), which resulted in CUTE unfolding unbounded loops, without ever attempting to cover any alternative branches in the code. Consider the example in Figure 10, taken from the `plot2d` program. The first predicate `if( to_x - x == 0 )` is satisfied by default (all primitive inputs are instantiated with equal values). The unbounded loop contained within the `if` statement ensures CUTE has an unlimited number of paths to explore before returning. Any code following the `if` statement is never explored, even though it makes up the bulk of the function.

The function `exitex` highlighted an interesting behaviour in CUTE. Its input space consists of two integer variables, one a global parameter and the other a formal parameter. The body of the function is shown in Figure 11.

In the first iteration, both `i` and `failed` will be 0. This means the function `_exit` is called with a value of 0, the status code for `EXIT_SUCCESS`. The `_exit` function terminates the running process and exits with the supplied status code. Since it indicates no error, it does not get caught by the registered signal handlers. CUTE however terminates before it can save its branch history

```
int exitex(int i) {
   if (failed != 0 && i == 0)
      i = failed;
   _exit(i);
   return 0;
}
```

Figure 11: Example of a function which causes CUTE to run until its iteration budget has been exhausted.

(of what has been covered) to a file for subsequent iterations. CUTE's constraint solver can easily find a solution to the constraint `failed != 0 && i == 0` and thus CUTE is left with an infinite amount of feasible branches and continues to run until its iteration limit is reached. However, the true branch of the `if` statement in Figure 11 will not be covered by CUTE in the process.

AUSTIN-specific issues. Search based testing is most effective when the fitness landscape used by an algorithm provides ubiquitous guidance towards the global (maximum) minimum. As discussed in Section 2.2, a landscape containing plateaus often reduces a guided search to a random search in an attempt to leave a flat fitness region. The 'shape' of the fitness landscape primarily depends on the fitness function used to evaluate candidate solutions (test cases).

Data dependencies between variables are a major contributor to plateaus in a fitness landscape, best illustrated by the flag problem [35]. While work has been done to tackle this problem, no standardized approach exists for including such information in the fitness computation. The fitness function used by AUSTIN does not explicitly consider any data dependencies.

The test subjects used during the empirical study contained many predicates exhibiting flag like properties, introducing spikes and plateaus into the fitness landscape. Consider the example shown in Figure 12 (taken again from `plot2d`), where the predicate depends on a function returned flag. When a flag only has relatively few input values which make it adopt one of its two possible values, it will be hard to find such a value [36].

For the above example, a random strategy of setting the input (and its members) to $NULL$ or non-$NULL$ will have a good chance of reaching the bulk of the body from `CPLOT_BYTE_MTX_Fill`. The code snippet was chosen not to illustrate the problem of function assigned flags, but to highlight another problem in AUSTIN's strategy. While symbolic information is collected across function calls, AUSTIN only computes intraprocedural control flow information. In order to take the false branch of the first `if` statement in `CPLOT_BYTE_MTX_Fill`, execution needs to follow the false branches of both conditionals in `CPLOT_BYTE_MTX_isNull`. The false branch shown in `CPLOT_BYTE_MTX_Fill` is not control dependent on the predicates in the function `CPLOT_BYTE_MTX_isNull`, and thus AUSTIN will not attempt to modify `dst`. AUSTIN's pointer rules (shown in Table 1) can only be applied to critical branching nodes (Section 2.2).

During the sample study, AUSTIN reached its runtime limit 9 times while attempting coverage of a function. No input domain reduction was used, and thus at times AUSTIN had to optimize large input vectors. In the worst case, it has to execute a function *at least* $2 * n + 1$ times to cover a target branch, where n is the size of the input vector. Previous work [34] has shown that reducing the search space for an AVM significantly improves its efficiency. When efficiency and a runtime limit are linked, it has an impact on the effectiveness of the search.

AUSTIN reported an abnormal termination of the program in 113 runs during the sample study. This does not mean having executed the `abort` function, but rather that the program

```
BOOL
CPLOT_BYTE_MTX_isNull(CPLOT_structByteMatrix *M)
{
  if( !M )
    return (1);
  if( M->data == ((void *)0) )
    return (1);
  return (0);
}

BOOL
CPLOT_BYTE_MTX_Fill(CPLOT_structByteMatrix *dst,
{
   ...
   if( CPLOT_BYTE_MTX_isNull( dst ) )
      return 0;

   //TARGET BRANCHES NEVER EXECUTED BY AUSTIN
}
```

Figure 12: Example used to demonstrate how function assigned flags cause problems for AUSTIN.

under test returned an unrecognised error code. Twenty three of these were caused by passing an unrecognised status code to an `exit` function, propagated through by inputs from the function under test. The remainder were triggered by `longjmperror`, which aborts a program. An execution environment is saved into a buffer by a call to `setjmp`. The `longjmp` command enables a program to return to a previously saved environment. The `longjmperror` is raised if the environment has been corrupted or already returned.

5. Tool vs Technique

It is important for different test data generation techniques, developed within different research communities, to be compared against each other. In order to perform such a comparison one needs tools which implement the different techniques. Yet, whenever one seeks to compare two different techniques, as is the case in this paper, one has to consider whether one is comparing the tools rather than the techniques implemented by the tools. The following section attempts to distinguish between shortcomings of a test data generation technique and its implementation, based on the findings in answering RQ3 (*Which types of program structure did each tool fail to cover?*).

5.1. Search Based Testing

Investigation of RQ3 revealed that search based testing suffered greatly from the 'flat fitness landscape' problem. When the fitness function produces the same fitness value for many different inputs (to the program), the fitness landscape will contain plateaus. These offer no guidance for the search algorithm and thus deteriorate a guided search into a random search. The problem is most acute in the presence of flag variables [37]; variables that can only take on one of two

values. Function returned flags (such as the example shown in Figure 12) were found to be particularly common. Many authors have worked on the flag problem [38, 39, 40, 41] in the context of search based test data generation. Wappler *et al.* [36] were the first to address the problem of function assigned flags by proposing a testability transformation [41]. Such transformations are independent of the underlying (search based) test data generation technique. While the testability transformation for function assigned flags has been shown to work on small examples, the feasibility of using such a transformation on medium to large scale, real world software has yet to be demonstrated. Furthermore, the transformation as proposed by Wappler *et al.* is not directly applicable to function assigned flags which depend on pointers, as is the case in Figure 12. This is because the testability transformation relies on similar branch distance calculations to the ones described in Section 2.2. However, as previously explained, branch distances over physical pointer addresses do not usually give rise to useful information for test data generation.

Another well known problem for search based testing are rugged fitness landscapes. In general the AVM works best on concave (up or down) functions with few local optima. While the random restart strategy described in Section 2.2 is designed to overcome local optima, the AVM will deteriorate more and more to a random (and thus an inefficient) search as the number of local optima increases. The fitness landscapes encountered for the different test subjects displayed some characteristics of a rugged landscape, though overall the problem of flat fitness landscapes was much more common.

In summary, more research is required to integrate landscape analysis into search based testing. Besides such landscape analyses, the search based community also needs to address the problem of generating input values for the following types:

- `union` - Currently there is no mechanism to track which member of a union is required in order to cover a particular target branch.
- `void*` - Void pointers are a commonly used feature in C programs, yet they present a big challenge to search based testing, because there is no analysis of the data types a void pointer may get cast to.
- `vararg` - Variable argument length lists present similar problems to void pointers for search based testing, in that required type information is missing.

5.2. Dynamic Symbolic Execution

One of the biggest problems with dynamic symbolic execution based techniques is deciding on a good path exploration strategy. In its default mode, CUTE uses an unbounded depth-first search. As the graphs in Figure 4 indicate (and the example in Figure 10 illustrates) this may be suboptimal. Choosing an adequate bound however, may not always be straightforward, especially if the unit under test contains many function calls. Further, Burnim and Sen [5] have shown that even a bounded depth-first path exploration is still inferior to other path exploration strategies. Other authors have also considered alternative approaches [42], though finding a (universally) good strategy remains an open problem.

A related problem is the well known path explosion problem in symbolic execution based techniques. CUTE reports the level of branch coverage it achieves, but in fact tries to explore all feasible execution paths (up to a specified length if one restricts its depth-first search). The problem of infinitely many paths has already been described. Many functions however contain a limited, yet very large number of feasible paths, and therefore the technique implemented in

CUTE cannot achieve high levels of coverage in a practical amount of time, even if all other implementation related details described in Section 4.3 could be addressed.

Finally, many of the problems listed in answer to RQ3 (*Which types of program structure did each tool fail to cover?*) could be solved by improving and refining the implementation of concolic testing in CUTE. In particular, if CUTE would be extended to handle more arithmetic operations, its effectiveness could be greatly increased. What impact a more advanced symbolic executor would have on the efficiency of CUTE remains an open question. Two other main deficiencies in CUTE remain. Support for strings and floating point operations. Other dynamic symbolic execution tools such as Pex [4] already support constraints over strings, and work is under way to develop constraint solvers for symbolic execution which support constraints over floating point variables [43].

6. Threats to Validity

Any attempt to compare two different approaches faces a number of challenges. It is important to ensure that the comparison is as fair as possible. Furthermore, the study presented here, as we are comparing two widely studied approaches to test data generation, also seeks to explore how well these approaches apply to real world code. Naturally, this raises a number of threats to the validity of the findings, which are briefly discussed in this section.

The first issue to address is that of the *internal validity* of the experiments, *i.e.*, whether there has been a bias in the experimental design that could affect the obtained results. One potential source of bias comes from the settings used for each tool in the experiments and the possibility that the setup could have favoured or harmed the performance of one or both tools. In order to address this, default settings were used where possible. Where there was no obvious default (*e.g.* termination criteria), care was taken to ensure that reasonable values were used, and that they allowed a sensible comparison between performance of both tools.

Another potential source of bias comes from the inherent stochastic behaviour of the meta-heuristic search algorithm used in AUSTIN. The most reliable (and widely used) technique for overcoming this source of variability is to perform tests using a sufficiently large sample of data. In order to ensure a large sample size, experiments were repeated 30 times. Due to the stochastic nature of some of the test subjects in the study, experiments were also repeated 30 times for CUTE, so as not to bias the results in favour of the AUSTIN tool.

A further source of bias includes the selection of the programs used in the empirical study, which could potentially affect its *external validity*; *i.e.*, the extent to which it is possible to generalise from the results obtained. The rich and diverse nature of programs makes it impossible to sample a sufficiently large set of programs such that all the characteristics of all possible programs could be captured. However, where possible, a variety of programming styles and sources have been used. The study draws upon code from real world open source programs. It should also be noted that the empirical study drew on over 700 functions comprising of over 14,000 branches, providing a large pool of results from which to make observations.

The data were collected and analysed in three different ways; taking into account coverage in the FUT only, interprocedural coverage and removing functions that CUTE could not handle from the sample. No matter which analysis was conducted, the results always showed a consistently poor level of coverage. Nevertheless, caution is required before making any claims as to whether these results would be observed on other programs, possibly from different sources and in different programming languages. As with all empirical experiments in software engineering, further experiments are required in order to replicate the results here.

7. Related Work

There have been several tools developed using Directed Random Testing. The first tool was developed by Godefroid *et al.* [2] during their work on directed random testing and the DART tool. Unlike CUTE, DART does not attempt to solve constraints involving memory locations. Instead, pointers are randomly initialized to either $NULL$ or a new memory location. DART does not transform non-linear expressions either and simply replaces the entire expression with its concrete value. Cadar and Engler independently developed EGT [3]. EGT starts with pure symbolic execution. When constraints on a programs input parameters become too complex, symbolic execution is paused and the path condition collected thus far instantiated with concrete inputs. Runtime values are then used to simplify symbolic expressions so that symbolic execution can continue with a mix of symbolic variables and constants. CREST [5] is a recent open-source successor to CUTE. Its main difference to CUTE is a more sophisticated, CFG based, path exploration strategy.

Pex [4] is a parametrized unit testing framework developed by Microsoft. Contrary to the majority of structural testing tools, it performs instrumentation at the .NET intermediate language level. As a result it is able to handle all 'safe' .NET instructions and can also include information from system libraries. Pex can be fully integrated into the Visual Studio development environment. Its tight coupling with the .NET runtime also allows it to handle exceptions, *e.g.* by suggesting guarding statements for objects or preconditions to functions.

Several tools have also been developed for search based testing. ET-S, developed by Daimler [21], uses evolutionary algorithms to achieve various coverage types, including path, branch and data flow coverage. IGUANA [44] is a tool developed for researchers, and incorporates different search approaches, as well as an API for the development of different objective functions. The eToc tool [22], implements an evolutionary strategy for JAVA classes. The tool evolves sequences of method calls in order to achieve branch coverage.

Xie *et al.* [45] were the first to combine concolic and search based testing in a framework called EVACON, which aims to maximize coverage of JAVA classes using both eToc and jCUTE, a JAVA version of CUTE [46].

There have been a number of previous empirical studies involving concolic and search based approaches.

Burnim and Sen [5] considered different search strategies to explore program paths in concolic testing and evaluated their findings on large open source applications including the Siemens benchmark suite [47], `grep` [48], a search utility based on regular expressions, and `vim` [49], a common text editor. An extended version of CUTE [31] has also been applied to the `vim` editor. Since its introduction, DART has been further developed and used in conjunction with other techniques to test functions from real world programs in an order of magnitude of 10,500 LOC [50, 32]. Concolic testing has also been used to search for security vulnerabilities in large Microsoft applications as part of the SAGE tool [33].

Studies in search based software testing have largely involved small laboratory programs, with experiments designed to show that search based testing is more effective than random testing [20, 21]. There are comparatively fewer studies which have featured real world programs; those that have, considered only libraries [22] or parts of applications [23] in order to demonstrate differences between different search based approaches.

The present paper complements and extends this previous work. It is the first to compare both approaches on the same set of non-trivial real world test subjects, without the need to modify

them in order to enable the application of test data generation tools. It is also the largest study of search based testing by an order of magnitude.

8. Conclusions

This paper has investigated the performance of two approaches to automated structural test data generation, the concolic approach embodied in the CUTE tool, and the search based approach implemented in the AUSTIN tool. The empirical study centred on five complete open source applications. The results show that both tools faced many challenges, with neither covering more than 50% of the branches in each application's code. A notable exception is the `plot2d` program for which AUSTIN achieves 77% coverage. The paper presents an analysis of the problems encountered by both tools, and aims to separate the shortcomings of each tool from its underlying technique.

9. Acknowledgments

Kiran Lakhotia is funded by EPSRC grant EP/G060525/1. Phil McMinn is supported in part by EPSRC grants EP/G009600/1 (Automated Discovery of Emergent Misbehaviour) and EP/F065825/1 (REGI: Reverse Engineering State Machine Hierarchies by Grammar Inference). Mark Harman is supported by EPSRC Grants EP/G060525/1, EP/D050863, GR/S93684 & GR/T22872 and also by the kind support of Daimler Berlin, BMS and Vizuri Ltd., London.

[1] K. Sen, D. Marinov, G. Agha, CUTE: a concolic unit testing engine for C, in: ESEC/SIGSOFT FSE, ACM, 2005, pp. 263–272.

[2] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, ACM SIGPLAN Notices 40 (6) (2005) 213–223.

[3] C. Cadar, D. R. Engler, Execution generated test cases: How to make systems code crash itself, in: Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings, Vol. 3639 of Lecture Notes in Computer Science, Springer, 2005, pp. 2–23.

[4] N. Tillmann, J. de Halleux, Pex-white box test generation for.NET, in: TAP, Vol. 4966 of Lecture Notes in Computer Science, Springer, 2008, pp. 134–153.

[5] J. Burnim, K. Sen, Heuristics for scalable dynamic test generation, in: ASE, IEEE, 2008, pp. 443–446.

[6] P. McMinn, Search-based software test data generation: A survey, Software Testing, Verification and Reliability 14 (2) (2004) 105–156.

[7] J. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, M. Shepperd, Reformulating software engineering as a search problem, IEE Proceedings — Software 150 (3) (2003) 161–175.

[8] M. Harman, The current state and future of search based software engineering, in: Future of Software Engineering 2007 (FOSE 2007), IEEE Computer Society, 2007, pp. 342–357.

[9] M. Harman, J. Clark, Metrics are fitness functions too, in: 10^{th} International Software Metrics Symposium (METRICS 2004), IEEE Computer Society Press, Los Alamitos, California, USA, 2004, pp. 58–69.

[10] O. Buehler, J. Wegener, Evolutionary functional testing of an automated parking system, in: International Conference on Computer, Communication and Control Technologies and The 9th International Conference on Information Systems Analysis and Synthesis, Orlando, Florida, USA, 2003.

[11] J. Wegener, F. Mueller, A comparison of static analysis and evolutionary testing for the verification of timing constraints, Real-Time Systems 21 (3) (2001) 241–268.

[12] B. Baudry, F. Fleurey, J.-M. Jézéquel, Y. L. Traon, From genetic to bacteriological algorithms for mutation-based testing, Softw. Test, Verif. Reliab 15 (2) (2005) 73–96.
URL `http://dx.doi.org/10.1002/stvr.313`

[13] S. Yoo, M. Harman, Pareto efficient multi-objective test case selection, in: ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, 2007, pp. 140–150.

[14] Z. Li, M. Harman, R. Hierons, Meta-heuristic search algorithms for regression test case prioritization, IEEE Transactions on Software EngineeringTo appear.

[15] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, R. S. Roos, Time aware test suite prioritization, in: International Symposium on Software Testing and Analysis (ISSTA 06), ACM Press, Portland, Maine, USA., 2006, pp. 1 – 12.
[16] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, C. J. Colbourn, Constructing test suites for interaction testing, in: Proceedings of the 25th International Conference on Software Engineering (ICSE-03), IEEE Computer Society, Piscataway, NJ, 2003, pp. 38–48.
[17] W. Miller, D. Spooner, Automatic generation of floating-point test data, IEEE Transactions on Software Engineering 2 (3) (1976) 223–226.
[18] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering 16 (8) (1990) 870–879.
[19] R. Pargas, M. Harrold, R. Peck, Test-data generation using genetic algorithms, Software Testing, Verification and Reliability 9 (4) (1999) 263–282.
[20] C. C. Michael, G. McGraw, M. Schatz, Generating software test data by evolution, IEEE Trans. Software Eng 27 (12) (2001) 1085–1110.
[21] J. Wegener, A. Baresel, H. Sthamer, Evolutionary test environment for automatic structural testing, Information and Software Technology 43 (14) (2001) 841–854.
[22] P. Tonella, Evolutionary testing of classes, in: Proceedings of the International Symposium on Software Testing and Analysis, ACM Press, Boston, USA, 2004, pp. 119–128.
[23] M. Harman, P. McMinn, A theoretical and empirical study of search based testing: Local, global and hybrid search, IEEE Transactions on Software Engineering.
[24] K. Lakhotia, M. Harman, P. McMinn, Handling dynamic data structures in search based testing, in: GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation, ACM, Atlanta, GA, USA, 2008, pp. 1759–1766.
[25] J. C. King, Symbolic execution and program testing, Communications of the ACM 19 (7) (1976) 385–394.
[26] A. Baresel, D. Binkley, M. Harman, B. Korel, Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004), ACM, Boston, Massachusetts, USA, 2004, pp. 43–52.
[27] P. McMinn, M. Harman, D. Binkley, P. Tonella, The species per path approach to search-based test data generation, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006), ACM, Portland, Maine, USA, 2006, pp. 13–24.
[28] G. C. Necula, S. McPeak, S. P. Rahul, W. Weimer, CIL: Intermediate language and tools for analysis and transformation of C programs, Lecture Notes in Computer Science 2304 (2002) 213–228.
[29] N. Williams, B. Marre, P. Mouy, M. Roger, Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis, in: EDCC, Vol. 3463 of Lecture Notes in Computer Science, Springer, 2005, pp. 281–292.
[30] C. Csallner, N. Tillmann, Y. Smaragdakis, Dysy: dynamic symbolic execution for invariant inference, in: ICSE, ACM, 2008, pp. 281–290.
[31] R. Majumdar, K. Sen, Hybrid concolic testing, in: ICSE, IEEE Computer Society, 2007, pp. 416–426.
[32] P. Godefroid, Compositional dynamic test generation, in: POPL, ACM, 2007, pp. 47–54.
[33] P. Godefroid, Random testing for security: blackbox vs. whitebox fuzzing, in: RT '07: Proceedings of the 2nd international workshop on Random testing, ACM, New York, NY, USA, 2007, pp. 1–1.
[34] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, J. Wegener, The impact of input domain reduction on search-based test data generation, in: ESEC/SIGSOFT FSE, ACM, 2007, pp. 155–164.
[35] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, M. Roper, Testability transformation, IEEE Transactions on Software Engineering 30 (1) (2004) 3–16.
[36] S. Wappler, A. Baresel, J. Wegener, Improving evolutionary testing in the presence of function–assigned flags, in: Testing: Academic & Industrial Conference, Practice And Research Techniques (TAIC PART07), IEEE Computer Society Press, 2007, pp. 23–28.
[37] D. Binkley, M. Harman, K. Lakhotia, FlagRemover: A Testability Transformation For Loop Assigned Flags., Transactions on Software Engineering and Methodology. To appear.
[38] A. Baresel, H. Sthamer, Evolutionary testing of flag conditions, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Vol. 2724 of LNCS, Springer-Verlag, Chicago, 2003, pp. 2442–2454.
[39] L. Bottaci, Instrumenting programs with flag variables for test data search by genetic algorithms, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002), Morgan Kaufmann Publishers, New York, 2002, pp. 1337–1342.
[40] R. Ferguson, B. Korel, The chaining approach for software test data generation, ACM Transactions on Software Engineering and Methodology 5 (1) (1996) 63–86.
[41] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, M. Roper, Testability transformation, IEEE Transactions on Software Engineering 30 (1) (2004) 3–16.
[42] T. Xie, N. Tillmann, P. de Halleux, W. Schulte, Fitness-guided path exploration in dynamic symbolic execution, in: Proc. the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009),

2009.
[43] B. Botella, A. Gotlieb, C. Michel, Symbolic execution of floating-point computations, Softw. Test, Verif. Reliab 16 (2) (2006) 97–121.
[44] P. McMinn, 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).
[45] K. Inkumsah, T. Xie, Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs, in: Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), 2007, pp. 425–428.
[46] K. Sen, G. Agha, CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools, in: Proceedings of the International Conference on Computer Aided Verification (CAV 2006), 2006, pp. 419–423.
[47] M. J. Harrold, G. Rothermel,
http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/Tools/subjects.
[48] G. grep, http://www.gnu.org/software/grep/.
[49] VIM, http://www.vim.org/.
[50] A. Chakrabarti, P. Godefroid, Software partitioning for effective automated unit testing, in: EMSOFT, ACM, 2006, pp. 262–271.