Proceedings Paper:
Kim, J., You, B., Kwon, M. et al. (2 more authors) (2017) Evaluating CAVM: A New Search-Based Test Data Generation Tool for C. In: Menzies, T. and Petke, J., (eds.) SSBSE 2017: Search Based Software Engineering. International Symposium on Search-Based Software Engineering (SSBSE 2017), 09-11 Sep 2017, Paderborn, Germany. Lecture Notes in Computer Science . Springer Cham , pp. 143-149.

https://doi.org/10.1007/978-3-319-66299-2_12

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

Junhwi Kim[1], Byeonghyeon You[1], Minhyuk Kwon[2], Phil McMinn[3], Shin Yoo[1]

[1] Korea Advanced Institute of Science and Technology, Republic of Korea
[2] Suresoft Technologies Inc., Republic of Korea
[3] University of Sheffield, UK

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, `CodeScroll`, 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 `CodeScroll` 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 `CodeScroll` nor `Austin` can, while also exclusively achieving the highest branch coverage for 20 of the studied functions.

1 Introduction

We introduce and evaluate `CAVM` (pronounced "ka-boom"), a new search-based test data generation tool for `C`. `CAVM` is based on the Alternating Variable Method (AVM) [7]: however, unlike the existing AVM-based test data generation tool `Austin` [6], `CAVM` 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 inputs (i.e., `char` arrays) for test data generation problems involving comparisons using the `strcmp` library function, using code rewriting.

We compare `CAVM` against a commercial test data generaton tool, `CodeScroll` (developed by Suresoft Technologies), and `Austin`, with respect to their relative effectiveness for `C` code involving dynamic data structures. The empirical evaluation studies small anti-pattern case studies, known to be challenging for `CodeScroll`, as well as real world open source and commercial code. The results show that our new algorithms, which we implemented into `CAVM`, can cover branches that neither `CodeScroll` nor `Austin` can.

2 `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 `CodeScroll` with a search-based software testing technique so that it can deal with challenging branches more effectively.

Extending the basic AVM for primitive types, CAVM adopts different local search strategies for each input type. For primitive data types, CAVM uses Iterated Pattern Search (IPS) [4,7]. In case of a struct type argument, CAVM applies AVM on each of its members: if the struct is nested, CAVM applies its AVM-based search algorithm recursively.

CAVM considers pointers to primitive types as arrays; CAVM initialises all pointers to NULL and applies IPS to each element of the current array, growing the size of the array by one if the search does not succeed. Note that the first "move" after failing to cover the given branch with NULL is to instantiate the pointer (i.e., *growing* it to a single element array) using a random value. CAVM grows dynamic data structures, such as linked lists or trees, by recursively growing nested pointers. For pointers to struct, if the current value is NULL, CAVM checks whether it can cover the current target branch simply by instantiating the pointer. CAVM randomly initialises primitive members of the instantiated struct. If the search does not succeed, CAVM subsequently tries to search for the values of the new instance (i.e., the members of the pointed struct) recursively. For more detailed description of CAVM and its algorithm, please refer to our technical report [5].

3 Experimental Setup

3.1 Subjects

Table 1 contains the list of subject functions that we study in this paper. The anti-pattern subject is a set of branches that CodeScroll 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 [7] 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. [3]. 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.

3.2 Configurations

We compare CAVM to Austin and CodeScroll based on the branch coverage they achieve. Since Austin and CAVM adopt stochastic approaches, we will report the average coverage over 20 runs. We only evaluate the deterministic heuristic of CodeScroll, and therefore do not repeat its runs.

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

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

[6] https://www.edg.com/c

Table 1: Subject C Functions Studied

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

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 [1] (i.e., coverage of branches that are not the target but nonetheless covered by a test case generated by a tool[7]). 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.

3.3 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[8] 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. CodeScroll only supports Microsoft Windows and consequently is executed on a machine with Intel Core i5-6600 3.9GHz and 16GB RAM running Windows 7. We allow the different hardware environments because we are only interested in achieved coverage and success rates.

4 Results

Table 2 contains the coverage results from 20 repetitive runs of Austin and CAVM, as well as single runs of CodeScroll. Note that the functions in decode.c

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

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

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

Function	Br.	CS	Austin		CAVM		Function	Br.	CS	Austin		CAVM	
			μ	σ	μ	σ				μ	σ	μ	σ
`AVMf`							`AVMf`						
`allzeros`$^{\square}$	6	0.00	0.00	0.00	**83.33**	0.00	`line`†	14	**100.00**	0.00	0.00	28.57	0.00
`calendar`*	46	**100.00**	0.00	0.00	0.00	0.00	`triangle`‡	16	**93.75**	0.00	0.00	89.06	5.32
`Antipatterns`							`decode.c`						
`case1`	4	0.00	**100.00**	0.00	**100.00**	0.00	`func1`	2	**100.00**	0.00	0.00	**100.00**	0.00
`case2`	4	75.00	**100.00**	0.00	**100.00**	0.00	`func2`	2	**100.00**	0.00	0.00	**100.00**	0.00
`case3`	2	50.00	**100.00**	0.00	**100.00**	0.00	`func3`	48	10.42	0.00	0.00	**29.90**	5.63
`case4`§	2	0.00	0.00	0.00	**100.00**	0.00	`func4`	14	21.43	0.00	0.00	**71.07**	6.34
`case5`	2	50.00	**100.00**	0.00	**100.00**	0.00	`func5`	14	**21.43**	0.00	0.00	0.00	0.00
`case6`	2	50.00	**100.00**	0.00	**100.00**	0.00	`func6`	16	18.75	0.00	0.00	**27.14**	9.44
`LinkedList`							`func7`	30	6.67	0.00	0.00	**11.56**	1.79
`delete`$^{\diamond}$	6	**100.00**	**100.00**	0.00	16.67	0.00	`func8`	6	50.00	0.00	0.00	**75.83**	12.65
`insert`$^{\diamond}$	8	87.50	**100.00**	0.00	50.00	0.00	`func9`	44	4.55	0.00	0.00	**69.66**	7.31
`modify`$^{\diamond}$	4	75.00	**100.00**	0.00	38.75	12.76	`func10`	28	7.14	0.00	0.00	**62.32**	10.20
`print_list`	2	**100.00**	**100.00**	0.00	**100.00**	0.00	`func11`	2	**100.00**	0.00	0.00	**100.00**	0.00
`search`	6	**100.00**	0.00	0.00	**100.00**	0.00	`func12`	4	25.00	0.00	0.00	**27.50**	7.69
`busybox-ls`							`func13`	4	50.00	0.00	0.00	**73.75**	5.59
`bold`	2	50.00	**100.00**	0.00	**100.00**	0.00	`func14`	2	50.00	0.00	0.00	**52.50**	11.18
`dnalloc`	2	**100.00**	**100.00**	0.00	**100.00**	0.00	`func15`	2	50.00	0.00	0.00	**97.50**	11.18
`fgcolor`	2	**100.00**	**100.00**	0.00	**100.00**	0.00	`func16`	12	8.33	0.00	0.00	**22.50**	18.56
`my_stat`	10	0.00	0.00	0.00	0.00	0.00	`func17`	4	25.00	0.00	0.00	**27.50**	11.18
`scan_one_dir`	16	**6.25**	0.00	0.00	0.00	0.00	`func18`	4	50.00	0.00	0.00	**64.17**	6.11
`BinaryTree`							`func19`	28	3.57	0.00	0.00	**8.75**	3.57
`inorder`	2	**100.00**	**100.00**	0.00	**100.00**	0.00	`func20`	8	87.50	0.00	0.00	**100.00**	0.00
`iter_inorder`	4	0.00	0.00	0.00	**100.00**	0.00	`func21`	4	**100.00**	0.00	0.00	**100.00**	0.00
`iter_search`	6	**100.00**	0.00	0.00	**100.00**	0.00	`func22`	18	**100.00**	0.00	0.00	**100.00**	0.00
`level_order`	8	62.50	0.00	0.00	**100.00**	0.00	Section 4/**RQ1** discusses the following issues.						
`postorder`	2	50.00	**100.00**	0.00	**100.00**	0.00	$\square$: indirect dependency. $*$: large search space.						
`preorder`	2	50.00	**100.00**	0.00	**100.00**	0.00	$\dagger$: low success rates. $\ddagger$: infeasible branches.						
`search`	6	**100.00**	0.00	0.00	**100.00**	0.00	$\diamond$: imprecise dependency analysis. $\S$: `strcmp`.						

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 `CodeScroll` 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):

(1) Indirect control dependency ($\square$): 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 [2]. This results in `CAVM` repeating random restarts.

(2) 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.

(3) 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.

(4) 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 `CodeScroll` cover all branches in `triangle`.

(5) Use of `strcmp` (§): `case4` in `Antipatterns` contains a call to `strcmp`, which neither `CodeScroll` nor `Austin` supports.

(6) 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, as it is not expressed as part of a nested structure. `CAVM`'s current control dependency analysis 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 the appropriate `else` structure), `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 `C` structural constructs is a part 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 `CodeScroll` nor `Austin` can. In particular, `Austin` has a significant limitation regarding pointer instantiation. The accompanying webpage[9] contains results about efficiency of `CAVM`, including the number of required fitness evaluations and the average wall clock execution time.

5 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, `CodeScroll`, 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 `CodeScroll`.

Acknowledgement. This work was supported by the ICT R&D program of MSIP/IITP [Grant No. R7117-16-0005: A connected private cloud platform for mission critical software test and verification].

References

1. 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)
2. 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)
3. Horowitz, E., Sahni, S., Anderson-Freed, S.: Fundamentals of Data Structures in C. W. H. Freeman & Co., New York, NY, USA (1992)
4. 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)
5. Kim, J., You, B., Kwon, M., McMinn, P., Yoo, S.: Evaluation of CAVM, Austin, and CodeScroll for test data generation for C. Tech. Rep. CS-TR-2017-413, School of Computing, Korean Advanced Institute of Science and Technology (2017)
6. 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)
7. 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

[9] `http://coinse.kaist.ac.kr/projects/cavm/`