Benchmarking Static Analyzers

Pascal Cuoq, Florent Kirchner, and Boris Yakobowski

CEA LIST*

1 Introduction

Static analysis benchmarks matter. Although benchmarking requires significant effort, it has driven innovation in many areas of Computer Science. Therefore this process and the underlying testcases should be carefully devised. However the problem that static analysis tackles—statically predicting whether a program is correct, or what it does when executed—is so hard that there exist no perfect oracle. For this and other reasons, there is little consensus on desirable and undesirable properties of a static analyzer. This article discusses some of these issues. Its examples involve the minutiae of C, but the principles should generalize to static analysis for most programming languages and, for some, to benchmarks for other partial solutions to undecidable problems.

2 Differing designs, differing goals

Benchmarks organizers should acknowledge that static analyzers use techniques that vary wildly in design and goals. To be representative and useful, the benchmark must be clear on the properties being tested.

At most one issue per test case When static analysis techniques are applied to C, they always only apply to defined executions. This is such a common restriction that it is not usually mentioned by those who make it. As corollary, a testcase should have at most one undefined behavior [4] (if the benchmark is about detecting undefined behaviors) or none (if the benchmark is about detecting another property, say, reachability of a specific label). Having more than one undefined behavior can cause mis-categorizations.

The reason is that it is allowable, and in fact desirable, for static analyzers to use so-called *blocking semantics* where the execution "gets stuck" on undefined behaviors. In fact, all tools use blocking semantics to some extent. In the example below, even high speed/low precision static analyses such as Steengaard's [7] exclude the possibility that `t` coincidentally ends up pointing to `c` because variable `d` happens to follow `c` in memory. Most analyses claim `s` and `t` do not alias. A tool might conclude that `*s == 3`, and that `reachable_p` is not called.

```
int c,  d,  *s = &c,  *t = &d - 1;
*s = 3; *t = 4;
if (*s == 4) reachable_p();
```

* Part of this work has been conducted during the ANR-funded U3CAT project.

Here, it is usual to consider that `reachable_p` is never called, because there are no defined executions that reach the call—although a concrete execution may reach it under circumstances outside the programmer's control. Technically, computing `t = &d - 1` is an undefined behavior [5, §6.5.6:8], and a tool is allowed to consider that execution stops there. Out of practical considerations, some may wait until `t` is dereferenced to assume execution gets stuck. In any case, the correct, consensual answer is that `reachable_p` is unreachable. The tricky situation is when there is no consensus on the semantics of the undefined behavior, as in the next paragraph.

Uninitialized variables as unknown values Some C static analyzers consider uninitialized variables simply as having unknown values. Developers of these tools may even have fallen into the habit of using this behavior as a feature when writing tests. This is their privilege as toolmakers. However, organizers should not assume that all analyzers take this approach. Other analyzers may treat an uninitialized access as the undefined behavior that it is [5, §6.2.4:5, §6.7.8:10, §3.17.2, §6.2.6.1:5] . Using the same "blocking semantics" principle that everyone uses, these analyzers may then consider executions going through the uninitialized access as stuck—and unfairly fail the test.

A better convention to introduce unknown values is a call to a dedicated function. Each toolmaker can then be invited to model the function adequately.

Well-identified goals The SRD test at `http://samate.nist.gov/SRD/view_testcase.php?tID=1737` implies that an analyzer should flag all uses of standard function `realloc`. The justification is that the `realloc`'ed data could remain lying in place in memory if the `realloc` function decided to move the block. However, absent any indication that the data is security-sensitive, the testcase is only measuring the ability of the analyzer to warn for any call to `realloc`: the standard does not specify any circumstances in which `realloc` is guaranteed not to move the pointed block (and having to sanitize the block before calling `realloc` defeats its purpose altogether). The C programming community does not universally consider `realloc` as a dangerous function always to be avoided. Thus an optimal analyzer with 100% marks on this particular problem might be rejected by programmers, who would consider it all noise and no signal.

A solution here is again to establish a simple convention to mark some memory contents as sensitive. If used consistently in all testcases in a benchmark, toolmakers can once and for all describe the relevant adjustments to make their respective tools conform to the convention. In general, the best way to make sure that a benchmark does not misrepresent the strengths and weaknesses of a tool is to include the toolmakers in the process [1].

Bugs in benchmarks We have said that organizers should not *willingly* incorporate undefined behavior in their testcases—unless they are testing the detection of this very defect. At the same time, we recommend organizers embrace the fact that they will *unwillingly* let a few undesirable bugs slip in. If some of the tested tools are described as warning for undefined behaviors, and if such warnings are emitted for constructs other than the known bug to detect, we recommend that a few of these reports be investigated, just in case.

It is very easy to slip. Canet et al [2] find a bug other than the bug to be detected in the Verisec benchmark [6]. The above-mentioned example 1737 from the NIST SRD contains an undefined behavior: the pointer returned by `realloc` is passed to `printf("...%s\n");` without having been tested for `NULL`. A tool may fail to warn for the `realloc` call being intrinsically dangerous, the debatable property being tested; piling irony upon irony, the same tool may warn about the nearby NULL dereference, a real issue but not the objective of the testcase—and this may be confused as a warning about `realloc`.

Categorizing analyzer outputs Most benchmarks categorize the tools' outputs as "positive" or "negative". This allows results to be synthesized into two well-understood metrics: precision and recall. However, from the point of view of the tool maker, this is frustrating. The differences between techniques are nowhere more visible than in the additional information they can provide. For instance, software model checking can optionally provide a concrete execution path. Abstract interpretation can guarantee that a statement is either unreachable or a run-time error (red alarms in Polyspace). Leaving room for some, but not all, of this information usually reveals the background of the organizers. Finally, since all techniques have their limitations, an "I have detected that the program is outside the scope of what I can do" category would be useful. It's a heartache for toolmakers to have to categorize this situation as either positive, negative or unknown, and it is misleading. It should be kept separate.

3 General benchmarking pitfalls

Do not make the problem too easy Benchmark testcases for static analyzers, in addition to staying clear of undefined behaviors, ought to have unknown inputs. Should they all terminate and come with fixed inputs, then a simple strategy to score on the benchmark is to unroll execution completely [4]. It is nice that a static analyzer is able to do that, but it does not measure how well it fares in actual use, predicting properties for billions of possible inputs at once. As an example of the danger here, the NIST Juliet suite contains testcases with mostly known inputs. A few have unknown inputs, but these are only boolean inputs that are still too easy to brute-force.

Licensing issues The necessity of obtaining a license for proprietary tools provides leverage to vendors: in addition to inquiring about the comparison, they may decline participation, or recommend a particular offering to be tested.

In contrast, the developers of Open Source tools are seldom given these opportunities, leading to two common challenges. One, the distribution of all-inclusive versions—a common behavior in academic communities less subject to marketing segmentation—require careful configuration, heightening the risk of misuse. This first pitfall can be avoided by contacting makers of Open Source analyzers and allow them to pick options for the task at hand: this is merely the equivalent of picking a tool from an artificially segmented commercial "suite".

Second, some academic licenses for commercial tools include restrictions on publication. One mitigation measure is to inquire whether the restrictions apply when the tool is anonymized [3] and to decide whether to anonymize and publish *before* results are known. Another is to extend the same privileges to all participants in the benchmark; since restrictions can go as far as a veto right on the resulting publication, organizers may well find this option unpalatable. In any case, restrictions that have been applied should be stated in the resulting publication as part of the testing protocol. We do not see quite enough of these caveats, even for comparisons that include commercial tools for which we know that licenses come with such strings attached [8].

4 Conclusion

Several C static analysis benchmarks already exist. It seems timely for this community to follow the same evolution automated proving has, and to move to larger—but still good-natured—competitions. But a large, recognized competition can only emerge if researchers with different backgrounds recognize themselves in it. To this end, basic principles must be agreed on. We propose some in this article. Consensus seems within reach: each of the benchmarks in the short bibliography applies most of the principles we recommend—but none apply them all.

This is a *discussion paper*. The conversation continues with more examples at `http://blog.frama-c.com/index.php?tag/benchmarks`.

Acknowledgments The authors thank Radu Grigore and Éric Goubault for their comments.

References

1. D. Beyer. Competition on software verification - (SV-COMP). In *TACAS*, 2012.
2. G. Canet, P. Cuoq, and B. Monate. A Value Analysis for C Programs. In *SCAM*, 2009.
3. G. Chatzieleftheriou and P. Katsaros. Test-driving static analysis tools in search of C code vulnerabilities. In *COMPSAC*, pages 96–103. IEEE Computer Society, 2011.
4. C. Ellison and G. Roşu. Defining the undefinedness of C. Technical report, University of Illinois, April 2012.
5. International Organization for Standardization. *ISO/IEC 9899:TC3: Programming Languages—C*, 2007. `www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`.
6. K. Ku, T. E. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In *ASE*, pages 389–392. ACM, 2007.
7. B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41. ACM, 1996.
8. M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *SIGSOFT*, 29(6):97–106, October 2004.