Testing static analyzers with randomly generated programs

Pascal Cuoq[1], Benjamin Monate[1], Anne Pacalet[2], Virgile Prevosto[1], John Regehr[3], Boris Yakobowski[1], and Xuejun Yang[3]

[1] CEA, LIST*
[2] INRIA Sophia-Antipolis
[3] University of Utah

Abstract. Static analyzers should be correct. We used the random C-program generator Csmith, initially intended to test C compilers, to test parts of the Frama-C static analysis platform. Although Frama-C was already relatively mature at that point, fifty bugs were found and fixed during the process, in the front-end (AST elaboration and type-checking) and in the value analysis, constant propagation and slicing plug-ins. Several bugs were also found in Csmith, even though it had been extensively tested and had been used to find numerous bugs in compilers.

1 Introduction

A natural place to start for industrial adoption of formal methods is in safety-critical applications [8]. In such a context, it is normal to have to justify that any tool used can fulfill its function, and indeed, various certification standards (DO-178 for aeronautics, EN-50128 for railway, or ISO 26262 for automotive) mention, in one way or another, the need to qualify any tool used. As formal methods make headway in the industry, the question of ensuring the tools work as intended becomes more acute.

This article is concerned with static analysis of software. A static analyzer can be formally correct by construction or it can generate a machine-checkable witness [2]. Both approaches assume that a formal semantics of the analyzed programming language is available. This body of work should not distract us from the fact that the safety-critical program is compiled and run by a compiler with its own version of the semantics of the programming language, and it is the executable code produced by the compiler that actually needs to be safe. Solutions can be imagined here too: the compiler could be a formally verified or verifying compiler based on the same formalization of the programming language as the static analyzer. However, it is not reasonable to expect that any safety-critical software industry is going to move *en masse* to such bundled solutions. Even the most enlightened industrial partners feel reassured if they can substitute one specific, well-delimited step in the existing process with a drop-in replacement based on formal methods [4]. In a context where we assume an early adopter

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

does not wish to change both compiler and verification process at the same time, how can we ensure the static analyzer being proposed agrees with the compiler already in use? We offer one partial solution to this problem: the compiler and the analyzer can be subjected to automated random testing.

Frama-C is a framework for analysis and transformation of C programs. It enables users to write their own analyses and transformations on top of provided ones. Several such custom analyses are used industrially [3] in addition to various R&D experiments [7]. Csmith [9] is an automatic generator of random C programs, originally intended for differential testing [6] of compilers. It has found bugs in all compilers it has been tried on, for a total of 400+ identified bugs. We used Csmith to test Frama-C. In this article, we report on the experiment's set-up and its results.

2 Testing Frama-C with random programs

Csmith generates programs that perform computations and then tally and print a checksum over their global variables. The programs contain no undefined or unspecified behavior, although they contain implementation-defined behavior. They are therefore deterministic for a given set of compilation choices. We defined several oracles for detecting that Frama-C wrongly handled a generated program (subsection 2.1). A few methodological remarks apply regardless of the oracle (subsection 2.2).

2.1 Testable Frama-C functionalities

Robustness testing Random inputs often uncover "crash bugs" in software. Any tool that accepts C programs as input can be tested for crashes with Csmith, discarding the output as long as the tool terminates normally.

Value analysis as C interpreter Frama-C's value analysis computes an over-approximation of the values each variable can take at each point of the program. Another abstract-interpretation-based static analyzer would work in a similar fashion, but make fundamentally different approximations, rendering any naïve application of differential testing difficult. Instead, a first way to test Frama-C's value analysis with Csmith is to turn Frama-C into a C interpreter and check its result against the result computed by the program compiled with a reference compiler.

To allow the value analysis to function as a C interpreter, we needed to make sure that all abstract functions returned a singleton abstract state when applied to a singleton abstract state. This was the hard part: since an abstract interpreter is designed to be imprecise, there are plenty of places where benign approximations are introduced. One example is `union` types, used in Csmith programs to convert between memory representations of different integer types. We had to improve the treatment of memory accesses to handle all possible partially overwritten values being assembled into any integer type.

Another difference between a plain and abstract interpreter is that a plain interpreter does not join states. The value analysis already had an option for improving precision by postponing joins, so this adaptation was easy. However, postponing joins can make fixpoint detection expensive in time and memory. Noting that when propagating singleton states without ever joining them a fixpoint may be detected if and only if the program fails to terminate, we disabled fixpoint detection. A Csmith program may fail to terminate, but for the sake of efficiency, our script only launches the value analysis on Csmith programs that, once compiled, execute in a few milliseconds (and thus provably terminate).

Printing internal invariants in executable C form The previous oracle can identify precision issues that cause the analyzer to lose information where it should not. It is also good at detecting soundness bugs, since the checksum computed by the program during interpretation is compared with the checksum computed in an execution. But a large part of the value analysis plug-in's nominal behavior is not tested in interpreter mode. In order to test the value analysis when used as a static analyzer, we augment it with a function to print as a C assertion everything it thinks it knows about the program variables. Partial information easily translates to C e.g. `x >= 3 && x <= 7` or, in the case of a relational abstract interpreter, `x - y <= 4`. To test, we analyze the program and obtain an assertion that is supposed to hold just before it terminates. The assertion is then inserted at the end of the program, and the program is compiled and run to confirm that the assertion holds.

The weakness of this oracle is that it cannot detect precision bugs. A precision bug makes the printed assertion weaker, but still true. Besides, the natural imprecision of static analysis may hide soundness bugs that would have been caught in interpreter mode. Therefore, both are indeed complementary.

Constant propagation A Frama-C program transformation plug-in builds on the results of the value analysis, replacing those expressions that only take one possible value throughout execution with their value. For testing this plug-in, the transformed program is compiled and run, and the computed checksum is compared to the checksum computed by the original program. This oracle tests the recording of value analysis results for further exploitation by other plug-ins, a feature that is not tested by the previous two methods.

Slicing Frama-C's slicing plug-in removes from an input program everything that does not contribute to the user-defined criterion. We sliced Csmith programs on the criterion "compute and print the final checksum." The slicing plug-in produces slices that can be compiled and executed, so for testing, we did that and compared the computed checksum to the original. This oracle cannot find slicing precision bugs where the plug-in includes unnecessary code, but it finds slicing soundness bugs where the sliced program computes a different checksum.

2.2 Methodological remarks

Observability By default, Csmith generates programs with command-line arguments and volatile variables. From the point of view of static analyzers, both command-line arguments and volatile variables are unknown inputs, for which all possible values are considered. Those initial imprecisions can snowball and absorb interesting (faulty) behaviors of the analyzers. We therefore used the Csmith options that disable these two constructs.

When this experiment took place, there was no automatic way to reduce a large Csmith-generated program triggering a bug into a small one triggering the same bug. Manual reduction took too much effort. The obvious solution was to make Csmith work harder to produce smaller test cases. Csmith does not have a single setting for generated program size, but several settings do influence average size, such as maximum number of functions, maximum size of arrays, and maximum expression complexity. We also let automatic scripts run longer, and then picked only the two shortest generated problematic programs out of the twenty found at each wave. Thus filtered, the test cases were 20KB on average, and acceptable for manual reduction. Once the stream of bugs dried up, we increased Csmith's settings again to default values and beyond, but no additional bugs were revealed: all the bugs identified with Csmith were found in the first phase.

Manual reduction When a Csmith program produces unexpected results, it may not be obvious where in the program lies the misinterpreted construct. In the case of Frama-C, one way we found to speed up bug identification was to add `printf()` calls at the beginning of each instruction block, so as to observe execution paths. This is not acceptable when manually reducing bugs in an optimizing compiler, because the calls interfere with optimizations. No such considerations apply to Frama-C's value analysis or plug-ins that exploit its results. The tested plug-ins handle statements one by one with little possibility of a bug being affected by interference between original and tracing statements.

Bug triage In a 300 kLOC piece of software such as Frama-C, the first step in fixing a bug is to classify it. Frama-C plug-ins build on the results of one another. When trying to make sense of a plug-in bug, it helps to be able to assume that the supporting plug-ins are reliable. We tested the plug-ins in bottom-up order so as to avoid bug reports that would be reclassified one or several times. Some parts of the framework can only be tested indirectly when their results are used by other parts. Thus, despite our efforts, a few bug reports had to be reclassified and looked at by more than one person.

3 Bugs found

The URL `http://j.mp/csmithbugs` lists bugs that were reported in Frama-C's bug tracking system. Oracles that use a reference compiler were applied with

GCC and Clang on IA32, PowerPC-32 and x86-64 targets, each time configuring the value analysis to simulate the corresponding target. Some bugs were indeed specific to big-endian platforms, or to the LP64 model (bug 785). One crash was identified and fixed (bug 715).

Value-analysis-as-interpreter testing revealed bugs in the front-end and in the value analysis itself, all of which could affect normal use as a static analyzer. An example of a bug in AST elaboration is the normalization into simple assignments of `x = long->access[path].bitfield = complex_expr;`. One constraint is that `long->access[path]` may contain arbitrarily complex expressions and should not be duplicated. Before the bug was fixed, this complex statement was normalized as `tmp = complex_expr; long->access[path].bitfield = tmp; x = tmp;`. Bug 933 explains why this is incorrect.

Testing the value analysis as a static analyzer found bugs related to alarm emission (bugs 715, 718, 1024), which was not tested at all in interpreter mode (no alarm is emitted while interpreting a defined program). Constant propagation testing revealed issues in program pretty-printing (bug 858). It was good to get these out of the way before testing the slicing plug-in. One slicing bug was found that could happen with very simple programs (bug 827). To occur, the other slicing bugs found needed the program to contain jumps in or out of a loop, or from one branch of a conditional to the other.

Unexpectedly, several bugs were also found by Frama-C in Csmith, despite the latter having been put to intensive and productive use finding bugs in compilers. Csmith is intended to generate only defined programs. The bugs found in Csmith involved the generation of programs that, despite being accepted by compilers, were not defined: they could pass around dangling pointers, contain unsequenced assignments to the same memory location, or access uninitialized members of unions. Generated programs could contravene [5, §6.5.16.1:3] either directly (`lv1 = lv2;` with overlapping lvalues `lv1` and `lv2`) or convolutedly (`lv1 = (e, lv2);`). Lastly, Csmith could generate the pre-increment `++u.f;` with `u.f` a 31-bit unsigned bitfield containing `0x7fffffff`. The bitfield `u.f` is promoted to `int` according to [5, §6.3.1.1] (because all possible values of a 31-bit unsigned bitfield fit in a 32-bit `int`). The increment then causes an undefined signed overflow. This last bug needs just the right conditions to appear. Narrower bitfields do not have the issue. Also, a 32-bit unsigned bitfield would be promoted to `unsigned int` (because not all its values would fit in type `int`) and the increment would always be defined [5, §6.2.5:9].

4 Related work, future directions and conclusion

Another recent application of fuzzing in formal methods is the differential testing of SMT solvers [1]. The originality of our experiment is that differential testing does not apply directly to static analyzers. New functionality had to be implemented in the value analysis both to make it testable as a plain C interpreter and to make it testable as a static analyzer. This was worth it, because the bugs found when misusing the value analysis as a plain interpreter also affected

nominal use. Besides, the resulting C interpreter is useful in itself for applications other than testing the value analysis.

It is not shocking that bugs were found in some of the Frama-C plug-ins that are already in operational use. Firstly, qualification of these plug-ins has not taken place yet. Secondly, some of the bugs found can only appear in a specific target configuration or in programs disallowed by industrial coding standards.

Fifty is a large number of bugs to find. It reveals the quantity of dark corners in the C language specification. That bugs were found in Csmith confirms this idea. Another inference is that correctly slicing a real world language such as C into executable slices is hard: it took 22 bug reports on the slicing plug-in itself to converge on an implementation that reliably handles Csmith-generated programs. The Frama-C developers were happy to be informed about every single bug. Almost all the bugs were obscure, which is as it should be, since Frama-C has been in use for some time now.

Csmith generates programs that explore many implementation-defined behaviors. Csmith testing not only uncovers bugs where either the reference compiler or the static analyzer disagree with the standard. It also checks that they agree with each other where the standard allows variations.

Our conclusion is that everyone should be testing their static analyzers with randomly generated programs. The Nitrogen Frama-C release can withstand several CPU-months of automated random testing without any new bugs being found. The development version is continuously tested, so as to find newly-introduced bugs as rapidly as possible.

References

1. Brummayer, R., Biere, A.: Fuzzing and delta-debugging SMT solvers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories. SMT '09, ACM, New York, NY, USA (2009)
2. Cachera, D., Pichardie, D.: Comparing Techniques for Certified Static Analysis. In: The NASA Formal Methods Symposium (NFM) (2009)
3. Delmas, D., Cuoq, P., Moya Lamiel, V., Duprat, S.: Fan-C, a Frama-C plug-in for data flow verification. In: ERTS^2 (2012), to appear
4. Delseny, H.: Formal Methods for Avionics Software Verification. Open-DO Conference, presentation available at `http://www.open-do.org/2010/04/28/formal-versus-agile-survival-of-the-fittest-herve-delseny/` (2010)
5. International Organization for Standardization: ISO/IEC 9899:TC3: Programming Languages—C (2007), `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`
6. McKeeman, W.M.: Differential testing for software. Digital Technical Journal 10(1), 100–107 (Dec 1998)
7. Pariente, D., Ledinot, E.: Formal Verification of Industrial C Code using Frama-C: a Case Study. In: FoVeOOS (2010)
8. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal methods: Practice and experience. ACM Computing Surveys 41(4) (2009)
9. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in C compilers. In: PLDI. San Jose, CA, USA (Jun 2011)