Analysing the Program Analyser

Cristian Cadar and Alastair F. Donaldson
Imperial College London, UK
{c.cadar, a.donaldson}@imperial.ac.uk

ABSTRACT

The reliability of program analysis tools is clearly important if such tools are to play a serious role in improving the quality and integrity of software systems, and the confidence which users place in such systems. Yet our experience is that, currently, little attention is paid to analysing the correctness of program analysers themselves, beyond regression testing. In this position paper we present our vision that, by 2025, the use of more rigorous analyses to check the reliability of program analysers will be commonplace. Inspired by recent advances in compiler testing, we set out initial steps towards this vision, building upon techniques such as cross-checking, program transformation and program generation.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Program analysis, testing, cross-checking, program transformations, program generators

Preface. It is the year 2025. FutureCorp, the private sector defence contractor to whom the US government has outsourced its management of nuclear weapons, has just had its missile control software hijacked by terrorists. It is only a matter of hours before Armageddon. The CEO of FutureCorp, Dr F. Methods, is incredulous: "This is impossible", he told one of our reporters. "We used program analysis to formally prove that the software was secure!". "Ah, Dr Methods," responded open-source developer Mr B. Door, "But did you check the analyser?"

1. INTRODUCTION

Automated program analysis techniques have tremendous potential to improve the reliability of the software systems on which our day-to-day life now depends. Yet for all the espoused benefits of program analysis by its proponents (authors of this article included), when asked the obvious question: "what do you do to check the analyser is correct?", with few exceptions the response of a tool developer is "we have a large test suite" at best, if not simply "well, it works on all the examples we have tried". This is despite the fact that program analysers are incredibly complex pieces of software.

Being unable to trust the correctness of a program analyser can seriously undermine its benefits. Imagine for example programmers having to constantly question the correctness of their debuggers—they would likely stop using them, reverting to less helpful but more accurate debugging aids such as print statements. Similarly, the utility of a verifier that claims to have proven the absence of a certain class of bugs, or of a testing tool that claims to provide inputs that achieve high coverage, is doubtful if the probability of implementation bugs in the tools themselves is high.

Our vision for 2025 is that continued advances both in the technology that drives program analysis, and the rigour with which program analysers are engineered, will provide a step change in the quality of analysers. Our inspiration comes from the world of compilers, whose correctness has attracted a great deal of attention [9–12, 14, 16, 19]. Broadly speaking, compiler quality is addressed by three mechanisms that go beyond manually-written test suites: (A) the "eat your own dog food" test, whereby a compiler is written in one of the languages it is capable of compiling, so that the compiler can be used to compile itself, (B) formal verification of compiler source code [10] or translation validation with respect to generated code [14] and (C) the application of automated testing methods, such as fuzzing [19] and metamorphic testing [9].

In the context of program analysers, the analogy of (A), the "dog food" test, would be for an analyser to analyse its own code base. While this may be possible in some instances, it is often difficult (e.g., the KLEE program analyser [5] is written in C++ and symbolically executes LLVM bitcode, thus could in principle be applied to itself by compiling the KLEE source code into LLVM bitcode, but lack of support for e.g., C++ libraries, makes this a difficult task in practice), limited in effectiveness (e.g., a buffer overflow checker could only find buffer overflow errors in its own code base), or simply impossible (e.g., it would be unreasonable to expect an analyser for GPU kernels, such as GPUVerify [1] to itself be implemented as a GPU kernel).

With regard to (B), formal verification, our view is that while these efforts are valuable and valiant, the human effort required to construct mechanised proofs of correctness for full-blown compilers and analysers is not feasible in general.

To appear in the International Conference on Software Engineering 2016, Visions of 2025 and Beyond Track. The definite version will be available at http://dx.doi.org/10.1145/2889160.2889206.

Our vision is to draw principally on the success of (C), automated compiler testing techniques, to yield practical methods for the analysis of program analysers. In this paper, we outline our ideas for *cross-checking* of analysis results (§2) and using *program transformations* as a basis for checking analysers (§3). Both approaches try to circumvent the *oracle problem*, which is particularly challenging in the context of program analysers, which one may regard as *non-testable programs* [17]. To some extent, these techniques depend on a source of interesting test programs. We believe that to enable thorough checking of specific analyses, advances are necessary in program generation methods that are *customisable*, generating programs that are relevant to the testing of a particular analysis, and *incremental*, enabling the form of programs to be modified as the capabilities of an analyser evolve (§4).

2. CROSS-CHECKING ANALYSERS

Cross-checking the results of an analyser with those of a similar tool, or performing internal consistency checks within a single analyser, are effective and often widely-available methods for bug-finding. Given the tremendous success that this approach has had in compiler testing—where it has found hundreds of bugs in real-world compilers such as `gcc` and `clang` [19]—we believe it is just a matter of time until we see a similar adoption in many different types of program analysis. The advantage of cross-checking analysers is that it does not require a specification, other than the fact that the analysers implement the same underlying analysis. For many types of analyses, there are often multiple implementations available—this is certainly the case for standard analyses such as dead code elimination or pointer aliasing, but also for more sophisticated ones like those based on symbolic execution or search-based testing. A method for testing refactoring tools applies this cross-checking idea [7]. Refactorings common to Eclipse and NetBeans are tested by applying each tool to a test program. Cases where the tools yield different refactored programs are flagged up for inspection, in case one of the tools has applied the refactoring incorrectly. Cross-checking has also been used to test constraint solvers [3], which underpin many symbolic program analysers.

The cross-checking approach could use either real programs, or small automatically-generated programs exhibiting relevant features (§4). For expensive analyses, the latter option might be the primary one: within a fixed time budget, two analysers might each only partially explore a complex, real-world program, making it hard to cross-check their incomplete results.

A challenge for cross-checking is making sure that the output of different analysers follows a standard specification—while this is often not the case, we hope that the prospect of improving analysers via cross-checking will provide the incentive for standardisation. Moreover, recent efforts on organising competitions for various kinds of analysers (e.g., SV-COMP [2]) are helping in this respect.

As for compilers, many implementation bugs are in the various optimisations that analysers perform. Therefore, comparing results with and without certain optimisations can cheaply and effectively find such bugs. Some ongoing experiments in the context of the symbolic execution engine KLEE have indeed revealed optimisation bugs.

For static program analysers, one can validate statically-inferred results against precise information obtained at runtime. For example, program invariants inferred by static analysis can be checked on a set of real executions. Such checks can be effective: a recent paper has shown that many pointer alias analysers incorrectly claim that two pointers never alias, when in fact they actually alias on real executions [18].

3. PROGRAM TRANSFORMATIONS AS A BASIS FOR CHECKING ANALYSERS

Creating relevant inputs that can be fed to program analysers in order to check that they are behaving correctly is challenging. While real programs are readily available, such programs unfortunately do not come with oracles. The cross-checking approach described in §2 can alleviate this problem, but it is not always applicable because different analysers might examine different parts of the program search space within a fixed time budget, and because when writing a brand new analysis for a specific program property, no other compatible analysis tools are available.

Instead, we propose the use of variations of mutation testing [8] and metamorphic testing [6] to test analysers on real(istic) programs. Starting from an existing program, one can perform program transformations to generate a slightly changed program with *predictably* different characteristics. The set of transformations, chosen according to the analysis of interest, should ideally: (1) lead to a better coverage of analysis features, e.g., by introducing new language features or new data structures, and (2) create programs for which it can be determined in advance whether a deviation in analysis results is expected compared with the original program.

Mutations. To achieve the latter goal, one could apply mutations that are either known to change expected analysis results, or to leave analysis results unchanged (even though they might otherwise change the semantics of the program). For instance, a points-to analysis should not be affected by a program modification that does not influence the conditions under which aliasing scenarios arise. In contrast, a conservative buffer overflow analysis should certainly be affected by a mutation that changes an in-bounds index variable to an index lying out of bounds.

Semantics-preserving transformations. A more general approach can introduce semantics-preserving program transformations (e.g., those performed by compiler optimisations, such as loop-invariant code motion, loop unrolling and inlining), which can have a significant impact on some types of analyses. For example, semantics-preserving transformations have been shown to lead to surprisingly large performance differences in dynamic symbolic execution [4]; semantics-preserving program transformations mimicking code changes by programmers have been used to evaluate code clone detection tools [15]; and removal of dead code is the basis of *equivalence modulo inputs* (EMI) testing, a recent technique proposed for testing compilers [9].

Recent work inspired by the EMI idea uses *opaque predicates* to inject "dead-by-construction" code into programs, and has proven effective in triggering miscompilations [11]. The idea is to equip an existing function with an extra boolean argument, `FF`, whose value is unknown to the compiler, and inject new code whose reachability is conditional on this argument:

if (FF) { /* *injected code* */ }

If *false* is provided as the value for `FF` at runtime, the program semantics should be unaffected by the injection. This means that if the original and injected program yield different results, the compiler must have miscompiled the program (under the assumption that the program is otherwise well-defined and deterministic). Reducing the injection to help identify the root cause of the bug is straightforward, by iteratively simplifying the injected code until further simplifications cause the mismatch to disappear.

This technique is applicable more generally, when the results of a static analysis pass are used to improve other static or dynamic analyses. For example, suppose that a symbolic execution tool can be accelerated via pointer aliasing information gathered by a static analysis: whenever a call through a symbolic function pointer is encountered, each potential function target (provided by the static analysis) is checked to see whether it satisfies the constraints gathered at that execution point. If so, a new path is forked to execute that function. As long as the static aliasing information is a conservative over-approximation, the symbolic execution tool should produce identical results; the more precise the alias analysis, the faster the symbolic execution stage.

Suppose we use the above strategy to inject code, constructed such that complex aliasing scenarios would arise if `FF` were *true*. Because the program should behave identically with or without the injection when *false* is passed as parameter `FF`, the symbolic execution tool should produce identical results. Because the static alias analysis has to work harder when analysing the injected program, due to its added complexity, bugs in the implementation of the alias analysis may be triggered. Differences in symbolic execution behaviour caused by wrong aliasing information would provide evidence that something is wrong with the static alias analysis.

Semantics-preserving transformations, as well as semantics-altering mutations under which the results of a particular analysis do not change, can both be seen as instances of *metamorphic testing* [6]. The key idea is to start with a program A (typically a real program), and then apply a transformation to generate a different program B, such that the properties of interest for the two programs are in a known relationship. Therefore, while it is unknown what the absolute results of program analysis should be for A or B, one can increase confidence in the analysis by checking that the results reported for A are related to the results reported for B in the expected manner. As a simple example, we may not know whether an analysis that reports the number of basic blocks in a program is correct, but it is clear that adding another basic block to the program should increase the reported value by one.

4. CUSTOMISABLE, INCREMENTAL PROGRAM GENERATORS

While starting with real programs and applying the techniques in §3 has the important advantage of using real or realistic programs, sometimes automatically generating programs that focus on specific features relevant to the analysis can be more effective at finding certain classes of errors. For instance, compiler testing has seen tremendous progress since the development of Csmith, a tool for automatically generating C programs without undefined behaviour [19].

Existing analyses can already benefit from program generators such as Csmith, but the resulting generated programs are not targeted toward checking specific analyses. We envision two ways of addressing this. One would be to specify, for a given analysis, a vocabulary of program features to be included in the generated program. For example, a buffer overflow checker would request programs with a higher density of array and pointer operations. A second idea is to exploit a form of *programming by example*. Starting from a set of example programs written by the developers that expose the kind of properties that the analyser should handle correctly, the generator would create a large number of similar programs, in order to stress-test the analyser and find mishandled corner cases. An approach along these lines has been applied in the context of testing refactoring tools using bounded-exhaustive testing, whereby the user specifies a number of *generators* for particular program constructs [7], in a manner such that generators can be composed to yield programs with a set of desired features.

An ambitious challenge is to make these program generators incremental with respect to the evolving code base of the analyser. Instead of generating programs in a manner that is oblivious to properties of the analyser under test, one would likely want to focus on testing newly added features. For example, if a tool for analysing OpenCL kernels adds support for reasoning about multiple command queues, one should be able to configure the program generator to primarily create programs with multiple command queues.

A key challenge faced by program generators like Csmith is ensuring that generated programs are free from undefined behaviour or, in the case of program analysis, that the only undefined behaviours are of the kind that the analysis under test is supposed to detect. This is because undefined behaviours in general allow a program analysis to output *any* result. For the case of arithmetic and bitwise operations, Csmith ensures definedness in a conservative manner by replacing all possibly unsafe operations with "safe math" versions, which yield some default value in the case where the result would be undefined. For instance, instead of issuing an unsigned division operation, `X / Y`, Csmith generates a guarded expression `Y == 0 ? X : X / Y`. This simple idea eliminates undefined behaviours, but makes the format of Csmith-generated programs rather prescribed, e.g., the division operator, `/`, only ever appears as the third argument to the ternary operator, `(·?·:·)`; the same holds for other operators with potentially undefined behaviour.

In the context of compiler testing, this prescribed form may prevent certain optimisations from triggering, thus denying them from being tested. In the more general context of program analysis, we believe that more sophisticated methods for limiting undefined behaviour in generated programs will be necessary. We propose leveraging mechanised programming language specifications during program generation (e.g., the Cerberus formalisation of C [13]), to search for programs that are guaranteed to be well-defined but are less prescribed than the programs generated by current techniques. A challenge here will be *efficient* program generation; the rate at which programs can be generated and executed is known to directly influence the rate at which compiler bugs are found using fuzzing; we hypothesise that this will also hold for testing program analysers.

5. RELATED WORK

Checking the correctness of program analysis tools is a natural thing to do, and our research agenda is inspired by our

own experience maintaining the GPUVerify [1] and KLEE [5] tools. Many of our ideas are based on well-known techniques such as crosschecking, mutation testing [8], metamorphic testing [6] and program fuzzing [19]. These have been broadly applied in numerous contexts, and we argue that they can be effectively adapted to test program analysers, as already demonstrated in prior isolated projects [7, 15, 18].

Compilers and interpreters can be seen as a particular type of program analyser, and more systematic effort has been put into testing and verifying them.

Compiler verification. The CompCert compiler showed that formal verification of a compiler for a real-world language is possible [10], but verification of optimisations remains extremely challenging, and the code base of a certified compiler is necessarily calcified by being coupled with a formal proof. We estimate the level of difficulty associated with formally verifying aspects of sophisticated program analyses to be comparable to that of verifying compiler optimisations. Further, program analysis tools must evolve in response to the growing number of contexts in which they are applied. We thus do not believe that formal verification of program analysis tools will be viable in general.

Compiler fuzzing. Random differential testing [12, 16, 19], where the results of multiple compilers are cross-checked against randomly generated programs, has been successful in uncovering bugs in numerous compilers. A limitation of current program generators such as Csmith [19] is that they can be hard to tailor and control, beyond tweaking the probabilities with which program constructs are generated and allowing specific classes of program features to be disabled. As discussed in §4, we believe that testing program analysers will require more flexible program generation strategies.

Program analysis competitions. Software verification tools targeting the C language compete in the annual SV-COMP competition, which has been running since 2012 [2]. This competition provides (a) a very large repository of benchmarks with known expected analysis results, (b) a large set of tools whose results over this benchmark suite are largely reproducible, and (c) a standard format for verification results, to allow comparison of tools. The set of submitted SV-COMP tools provide an army of analysers that can be used to cross-check a new analyser (§2). As well as forming a large regression suite, the benchmarks provide a rich source of examples on which program transformation-based testing can be applied (§3). A limitation of SV-COMP is that it is currently mostly restricted to tools that check whether assertions in C programs can fail.

6. CONCLUSION

The reliability of program analysis tools is clearly important if such tools are to play a serious role in improving the quality and integrity of software systems. We argue that a step change in the way analysers are tested is needed, and could be made possible by adapting well-known techniques such as crosschecking, mutation testing, semantics-preserving program transformations and metamorphic testing. To this end, developers of program analysis tools could agree on standardised output formats, and share standard test suites and libraries of program transformations specific to their analysis. More research into flexible, incremental program generators would also help realise this goal.

7. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and Jeroen Ketema, Tomasz Kuchta, Daniel Liew, and Hristina Palikareva for their comments on this paper, and EPSRC for supporting this research via EP/L002795/1.

8. REFERENCES

[1] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, and J. Wickerson. The design and implementation of a verification technique for GPU kernels. *TOPLAS*, 37(3):10, 2015.

[2] D. Beyer. Software verification and verifiable witnesses (Report on SV-COMP 2015). In *TACAS'15*.

[3] R. Brummayer and A. Biere. Fuzzing and delta-debugging SMT solvers. In *SMT'09*.

[4] C. Cadar. Targeted program transformations for symbolic execution. In *ESEC/FSE NI'15*.

[5] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI'08*.

[6] T. Chen, S. Cheung, and S. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Hong Kong University of Science and Technology.

[7] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *ESEC/FSE'07*.

[8] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *TSE*, 37(5):649–678, 2011.

[9] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *PLDI'14*.

[10] X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

[11] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *PLDI'15*.

[12] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10:100–107, 1998.

[13] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. Watson, and P. Sewell. Into the depths of C: elaborating the de facto standards. In *PLDI'16*.

[14] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS'98*.

[15] C. Roy and J. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Mutation'09*.

[16] F. Sheridan. Practical testing of a C99 compiler using output comparison. *SPE*, 37(14):1475–1488, 2007.

[17] E. J. Weyuker. Pseudo-oracles for non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[18] J. Wu, G. Hu, Y. Tang, and J. Yang. Effective dynamic detection of alias analysis errors. In *ESEC/FSE'13*.

[19] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI'11*.