Comparing Model Checking and Static Program Analysis: A Case Study in Error Detection Approaches

Kostyantyn Vorobyov
Bond University, Gold Coast, Australia

Padmanabhan Krishnan
Bond University, Gold Coast, Australia

Abstract

Static program analysis and model checking are two different techniques in bug detection that perform error checking statically, without running the program. In general, static program analysis determines run-time properties of programs by examining the code structure while model checking needs to explore the relevant states of the computation.

This paper reports on a comparison of such approaches via an empirical evaluation of tools that implement these techniques: CBMC – a bounded model checker and Parfait – a static program analysis tool. These tools are run over code repositories with known, marked in the code bugs. We indicate the performance of these tools and report on statistics of found and missed bugs. Our results illustrate the relative strengths of each approach.

1 Introduction

The safety of software can be a mission critical issue in computer systems. A significant number of program errors are still found even in software that has been thoroughly tested. Such flaws could lead to a situation which severely compromises the system's security. This is especially the case for low level system software (such as operating systems), which are usually implemented in the C programming language.

While using C has a number of benefits (including access to system resources, relatively small size of code) there are many sources of potential flaws. These errors can increase the level of vulnerabilities thereby exposing systems to a significant number of attacks, which often leads to very serious consequences, including inconsistent and unpredictable program behaviour, undetectable disclosure of confidential data, run-time errors and system crashes. Consequently there is a necessity in effective detection of such errors.

There are many general ways to tackle the problem of bug detection, including manual code inspection, testing [25], static and run-time analyses [24], model checking [23] or a combination of these techniques.

While testing remains the principal approach to detect bugs, recent advances have made it possible to consider other techniques such as static program analysis and model checking for large systems.

Static program analysis is the automatic determination of run-time properties of programs [12], which considers run-time errors at compilation time automatically, without code instrumentation or user interaction. Static analysis builds an abstract representation of the program behaviour and examines its states. This is done by the analyser maintaining extra information about a checked program, which has to be an approximation of the real one. Static analysis may result in sound analyses, that is, if a program is declared to be safe, it is indeed safe. But owing to the nature of the approximations to analysis may report errors that actually do not exist. Analysis that is not sound (especially involving pointers in C) is also used [20].

Model checking [23, 11] on the other hand computes the run-time states of the program without running the program. The generated run-time states are used to check if a property holds. If the program has a finite number of states, it is possible to do an exhaustive analysis. The program is said to be verified if program behaviour satisfies all properties in the specification. If such an execution path, that would lead to a violation of any of the properties, can be found, the program is said to fail verification. The result returned by a model checker is either a notion of a successful verification, or a counterexample – an execution path that violates a given property. If the program is not finite state, certain types of approximations are necessary, or a model-check may not terminate. For instance one can have bounded model checking [5, 9], where the number of states explored are bounded. Such approach is not sound – as a bug can exist beyond the bound. However, all bugs within the bound can be identified.

Based on the model checking and static program analysis characteristics, one usually makes the following general observations: if one examines running time and resource consumption, model checkers will be a more expensive approach in bug detection than static program analysis. This has been the motivation to develop static analysers for large code bases [8]. However static analysis is not as accurate as model checking. So a model checker should be able to produce more precise answers. An interesting question is does the accuracy of the results from model checking justify the extra resources it consumes.

Engler and Musuvanthi [15, 14] demonstrate results that dispel some of these common beliefs related to model checking vs static analysis. Our work is similar (i.e., to check whether these beliefs hold and if so under what circumstances) although there are a few key differences. One is that that we use standard code bases with known bugs. Furthermore we do not explicitly write any specification for model checking – we only check for a single property, viz., buffer overflow. Thus our comparison is about the tools for the buffer overflow property. We also compare the behaviour on precision and performance.

In this paper we present our results comparing static analysis and model checking on benchmarked (i.e., with known marked bugs) code bases. Our aim is to find out if it is possible to determine when to use static analysis and when to use model checking. Firstly, we need to determine whether the two approaches are comparable, and if so, establish the way it may be performed and identify the specific comparison criteria.

2 Experimental Setup

There are many possible tools that we could have chosen including lint (a consistency and plausibility static bug checker for C programs) [18], PolySpace (a static analysis tool for measuring dynamic software quality attributes) [13] Splint (an open source lightweight static analysis tool for ANSI C programs) [16], BLAST (a model checker that utilizes lazy predicate abstraction and interpolation-based predicate discovery) [4], SLAM (a symbolic model checking, program analysis and theorem proving tool) [3], SATURN (A boolean satisfiability based framework for static bug detection) [27].

We choose Parfait and CMBC for this comparison. The choice of these tools were dictated by their availability and prior uses for large programs.

CBMC is a bounded (via limiting the number of unwinding of loops and recursive function call depths) model checker for ANCI C programs [10]. CBMC can be used for the purposes of verification of array bounds, pointer safety, exceptions and user-specified assertions. Additionally, the CBMC tool checks if sufficient unwinding is done [1], ensuring that no bugs exist beyond the bound. If the formula is verified but cannot prove that sufficient unwinding has been performed the claim fails verification. The program is verified when all identified formulas are verified and no unwinding violations exist.

Parfait [8, 6] is a static bug checker for C and C++ source code designed by Sun Microsystems Laboratories (now Sun Labs at Oracle). This tool was created specifically for the purpose of bug checking of large source code bases (millions of lines of code). Parfait operates by defining an extensible framework composed of layers of program analyses. Initially Parfait generates a sound list of bugs – the one that contains all possible program errors, including such that do not exist. Parfait iterates over this list, applying a set of different analyses, accepting a bug as a real one and moving it to the set of real detections, or rejects it as non-existent. Analyses are ordered from the least to most expensive, and are applied to ensure that bug statements are detected in the cheapest possible way. We should note that Parfait is different to other static program analysis tools and was designed to have a low false positive rate (e.g. report only bugs that are proven to be ones during the analysis.)

Cifuentes et al [7] have identified common pitfalls and provide a framework to compare error checking tools. Using this framework, we focus on *precision* which is the ratio of bugs that are reported correctly to bugs that are reported but do not exist and *scalability* – ability of to tool to produce reports in timely and efficient manner.

In our comparison we measure precision by classifying the results from the tools into the following 3 categories:

- **True Positive** – reports that are correct (i.e. existing bugs)
- **False Negative** – undiscovered bugs
- **False Positive** – reports that are incorrect (i.e. non-existing bugs) where non-existent bugs are reported as bugs

We also identify and report on *True Negative* results, defined as the difference between the number potential bugs identified by the tool and real bugs, to better demonstrare differences in approaches.

It is important to enable a fair evaluation, because different tools target specific types of bugs. For instance, Parfait performs a memory-leak analysis, not implemented in CBMC. In order to avoid such issues, we limit our comparison to *Buffer Overflows* [2], which is the most common type of memory corruption and security exploit. A buffer overflow occurs when data is copied to a location in memory that exceeds the size of the reserved destination area. Other types of bugs that may be present in the code are not considered.

For the effective evaluation of the effectiveness of the tools we use a benchmarking framework – a repository with known bugs which are documented. This allows us effectively test the performance of the tools [21] and map detections to false positive, false negative and true positive results. For this evaluation we used the BegBunch [7] – benchmarking framework developed by Sun Labs at Oracle. The BegBunch framework includes four suites(code repositories) with bugs marked in the source code according to their type, which allows filtering out the types of errors not relevant for this comparison. In our comparison we consider the code from the *Iowa* [22], *Cigital* and *Samate* [19, 26] suites from the BegBunch and only used programs that contain buffer overflows. To be precise in detection of this type of bug and overcome issues related to differences in analyses of the tools, we limit our search to small sized programs that contain one or two buffer overflows. Thus we do not use code from *Parfait-Accuracy* – the fourth BegBunch suite, which programs often contain bugs of different types.

Although Parfait and CBMC are both used to identify bugs they differ in the reporting of the results. Parfait reports include found bugs including extra information of bug type (e.g. buffer overflow, memory leak etc) and bug origin. Additionally Parfait is integrated into BegBunch framework and is able to reveal full statistics over the set of programs from BegBunch, including true positive, false positive, false negative detections and the overall running time of the tool. CBMC, on the other hand, being a model checker reports only the notion of successful verification or a counterexample, if verification fails. Furthermore it stops when the first error is found. Thus we need to map the results produced by CBMC into the established comparison criteria, viz., false positive, true positive and false negative bug detections and determine the overall running time. Additionally to that we need to collect memory usage statistics for both tools.

Our experimentation process is as follows:

- Run Parfait and collect the results. As the BegBunch was initially designed for the purpose of testing Parfait it reports relevant statistics (i.e. false positives, false negatives, true positives and overall running time). We only needed to measure the memory consumption and calculate true negatives.
- For every program, identify the set of properties (claims in CBMC terminology) that CBMC intends to check.
- Conduct a separate CBMC run for each property and collect the results. For each run measure the time consumption and memory usage.
- Classify the violated properties into true positive, false positive and false negative detections.

In this experimental set up we would expect CBMC to have a greater resource usage for verification of the same code, when compared with Parfait.

Parfait can be expected to generate a low false positive error rate. CBMC is a sound tool only if enough unwinding of loops, function calls are done. So we could get a few false negatives. But as CBMC does exhaustive state exploration we can expect no false positive detections for CBMC.

2.1 Configuration of the tools

Parfait invocation was performed via the BegBunch interface. We have specified the lists of programs to run as a plain text file, where each benchmark name appears on the new line (a BegBunch inbuilt feature for Parfait runs with multiple programs). Reports produced by the framework include relevant bug statistics in terms of running time and detection type (true positive, false positive,false negative) for all applied analyses. We have have filtered out the non buffer overflow detections on the stage of BegBunch reporting. Initially BegBunch was created for the purpose of testing Parfait, therefore no issues were expected or encountered while running Parfait with the BegBunch code.

A feature of CBMC is that it can be customised to focus only on certain types or errors. Given our experimental framework we were interested only in array bounds, pointer checks and arithmetic over and underflow checks. These were specified using the `--bounds-check`, `--pointer-check`, and `--overflow-check` flags. CBMC also performs a simple form of slicing, where it removes assignments unrelated to the property being checked.

As formerly mentioned, in order to perform effective and fair comparison, the CBMC output (a set of violated claims) should be interpreted into false positive, false negative and true positive bug detections. We have processed every claim in a specific manner examining the trace of each particular violated property, creating a simplified claim, that would contain information about bug origin in terms of program line and function name where it has occurred.

In order to gather all the necessary statistics on all the examined programs, scripts that automated the runs of CBMC were created.

We now describe the instrumentation of the run-time measurement in CBMC. Due to the specifics of verification with CBMC (claim-by-claim basis to force CBMC to find all bugs), the time evaluation might not be considered as *completely fair* due to the overlapping of the same program structure generated in memory for every claim (considering that every claim evaluation is a separate process). However, taking into account the speed of SAT solving (much slower than structure generation) the overall running time (total time of consequent runs of all claims) was used. The program analysis part was amortised over all the claims and did not contribute a significant value.

3 Results

Now we present the reports produced by CBMC and Parfait over Iowa(Table 1), Samate (Table 2) and Cigital(Table 3) suites. The aggregated results for all suites are presented in Table 4. The results compute true positives, false negatives and false positives as well as the running time and highest memory peak per suite and overall. We discuss tool's performance false negative and true positive results in each of the sections and false positive results in Section 3.4.

We also report on true negative results for Parfait. This is because Parfait performs its analysis in two steps – searches for all possible errors and filters out false positives. We acknowledge that such results are intermediate, are not reported by Parfait by default and should not be used in calculation of Parfait's precision level. We include true negatives in order to demonstrate the differences in model checking and static program analysis approaches: where CBMC performs exhaustive verification of program states within a bound and stops, if a violation is found, whereas Parfait considers potential bugs and analyses them one by one to determine whether the bug should be accepted and reported as true positive or rejected as a false negative. True negatives better demonstrate the precision of a program static analysis tool used and give a broader view on false negatives and false positives of Parfait. Not just as bugs missed, correctly and incorrectly reported, but as bugs that were correctly rejected (true negatives), wrongly rejected (false negatives), wrongly accepted (false positives), correctly accepted (true positives).

We do not include such a measure for CBMC. Theoretically, the closest approximation to a true negative result for this tool can be the number of claims that CBMC checks. However, in practice, program properties from the model checking point of view do not directly correspond to the bugs in static program analysis approach – several violated properties might correspond to a single bug and vice versa (we have discussed the issue of mapping claims to bugs in the previous section and we choose to include such results for Parfait as we perform such a mapping and looking at the results from *bugs* perspective).

Additionally we report the *accuracy* rate for every suite and overall. Such calculations were done using the formula derived by Williams and Heckman [17]. Accuracy is calculated based on false positive (FP), false negative (FN), true positive (TP) and true negative (TN) results for the tools

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

We do not use the notion of true negative result in our calculation of accuracy, because it can not be clearly defined for CBMC and would not be a fair measure of comparison:

$$Accuracy = \frac{TP}{TP + FP + FN}$$

3.1 Iowa results

Iowa Bugs:421 Benchmarks:415		
	CBMC	Parfait
True Positive	413	274
False Positive	0	0
False Negative	8	147
True Negative	-	60
Running Time	18:40:59	00:00:47
Memory Peak	2.497 Gb	6712 Kb
Accuracy	98%	65%

Table 1: Iowa Verification Results

We have used 415 programs from Iowa suite, where the largest program is 136 lines, the smallest program is 43 lines. The average code size is 94 lines. For the Iowa code base CMBC is more precise. It missed only 8 bugs which is 1.87 percent of all bugs. While Parfait found about 69 percent of all bugs. However CBMC took over 18 hours to compute the result. The acceptability of this time is questionable. The memory peak is more acceptable given current machine configurations.

3.2 Samate results

We have considered 997 programs from Samate code base, where the largest program is 618 lines, the smallest program is 6 lines. The average code size is 18 lines. In the Samate code base CMBC was also more precise. It missed 35 bugs which is about 3 percent of bugs. Parfait on the other hand had accuracy rate of 84% comparing to 97% by CBMC, and correctly rejects 290 bugs against 167 false negatives.

CBMC had a worse false negative rate and has taken considerably less time for verification and lower memory consumption comparing to the results of the Iowa

Samate Bugs:1033 Benchmarks:997		
	CBMC	Parfait
True Positive	998	866
False Positive	0	0
False Negative	35	167
True Negative	-	290
Running Time	01:16:18	00:01:26
Memory Peak	307.388Mb	6748 Kb
Accuracy	97%	84%

Table 2: Samate Verification Results

suite. Although the Samate test suite contains twice as many programs, they clearly differ in complexity. Parfait has shown better results for false negative detections, staying at approximately 16 percent of precision which is considerably better than for previous suite.

3.3 Cigital results

Cigital Bugs:11 Benchmarks:11		
	CBMC	Parfait
True Positive	11	0
False Negative	0	11
False Positive	0	0
True Negative	-	1
Running Time	00:01:50	00:00:01
Memory Peak	94.668 Mb	4372 Kb
Accuracy	100%	0%

Table 3: Cigital Verification Results

We have considered 11 programs from Cigital code base, where the largest program is 18 lines, the smallest program is 5 lines. The average code size is 8 lines. For Cigital CBMC is complete and sound (no bugs missed, no false positives detected). CBMC shows good running time and low memory consumption, which is still much greater than Parfait's. However it should be accounted that even for such running time Parfait misses all bugs and therefore accuracy for Parfait and CBMC significantly differ – 0% against 100%.

3.4 Overall results

Overall results show that model checking approach has been much more expensive in terms of running time (over 19 hours) and memory peak (approximately 2.5 Gb). Such results are gathered over a large amount of small sized programs which indicates that running CBMC over a really large code base could be extremely time and resource consuming. However CBMC shows very good accuracy rate – 97% overall, and finds approximately 19% more bugs. Both tools have reported a zero rate over false positives, which is common for model checkers and specific to Parfait design. As formerly noted Parfait was designed to produce a small false positive rate for large scale programs. Which is why the results for small sized programs hold at the very low rate – zero in our case.

Overall Bugs:1465 Benchmarks:1423		
	CBMC	Parfait
True Positive	1422	1140
False Negative	43	325
False Positive	0	0
True Negative	-	351
Running Time	19:59:07	00:02:14
Memory Peak	2.497 Gb	6748 Kb
Accuracy	97%	77%

Table 4: Overall Verification Results

4 Discussion and lessons learned

The ratio in bug detection of the tools depend on the level of approximation of a model to a real program. And whilst static program analysis tools simplify such a model for better scalability characteristics and ease of analysis, model checkers try to create an equivalent model with the exact program properties, inheriting code complexity and thus extending resource consumption, but achieving higher precision. This difference in approaches is effectively demonstrated by the performance of Parfait and CBMC and the rate of true positives – model checker has introduced significantly better accuracy, as in 97% of true positive detections with CBMC against 77% with Parfait, however was extremely slow and resource demanding hitting the limit of 4 Gb per process for a 32 bit system. We report the highest memory usage of CBMC of 2.497 Gb, as this was the greatest memory peak of a workable verification. We have learned that insufficient unwinding was one of the reasons for false negative detections for CBMC. In our approach, in cases of insufficient system resources we were limiting the verification process to a performed finite run with the largest possible resource allocation. Another reason lies in difference of internal libraries of CBMC and Parfait. In our approach we worked with the bare bone systems and thus automatically treated undetected bugs that were occurring due to non-included system library code as false negatives. In case of static program analysis tool, such results are, of course, in direct relation to a complexity of an abstract model, which is simpler comparing to CBMC.

During such a comparison we have learned that there are certain limitations in the approach we have taken, which do not allow examine the current question in full depth. Such issues include:

Differences in functionality of internal libraries – neither of the tools we used performs dynamic analysis and thus rely on the source code of inbuilt C standard library, which in case of CBMC does not fully include usual functionality. This often results in false negative results, however may be altered by including needed source code.

Definitions of analyses – bug, that one tool can identify as buffer-overflow, another may refer to as an array-bounds-violation, similarly, as one may not be targeted for certain types of errors the other is. Thus, it needs to be decided whether a non-found bug is supposed to be one or it is a false negative detection. Prior to this we have limited our search to small sized programs, as we could not reliably determine types of bug detections that occur along with buffer-overflows. Programs we used usually contained one or two buffer overflows only.

Despite discussed limitations, based on the received results, we may conclude that in buffer overflow analysis static bug checkers can get close to model checkers narrowing the trade-offs of speed/accuracy to the ratio of false negative detections. However, gathered results effectively show that given the running time and the resource usage of CBMC it would not be possible to model check the code base of the size of an operating system, which means that in this respect static program analysis generally wins. Authors may suggest that in large code bases both techniques may be successfully used together to balance the speed/precision factors.

5 Conclusion

This paper has described our experience in comparing two approaches in error detection – static program analysis and model checking to answer the question of trade-offs between these techniques commonly presented as *slow and precise* for model checking and *fast and inaccurate* for static analysis. We have taken an empirical approach in evaluation via a comparison of the results produced by the tools that implement these methods. Our choice of the tools was dictated by their availability and prior use to large code bases. These results are in some variance with the results reported by Engler and Musuvathi [14].

Our evaluation of a static program analysis tool and a model checker has demonstrated common differences in two approaches in terms of running time, memory consumption and ratio missed and discovered bugs. We may conclude that ratio of false positives – incorrectly identified bugs, in some cases of static program analysis might be effectively reduced to the bare minimum.

Acknowledgements

We thank Cristina Cifuentes and Nathan Keynes for their sustained support during this project. They helped us understand Parfait and its application to the various code bases.

References

[1] *The CPROVER user manual, SatAbs – Predicate Abstraction with SAT, CBMC – Bounded Model Checking*. http://www.cprover.org/cbmc/doc/manual.pdf.

[2] BALABAN, M. Buffer overflows demystified. http://www.enderunix.org/documents/eng/bof-eng.txt.

[3] BALL, T., AND RAJAMANI, S. K. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software* (New York, NY, USA, 2001), Springer-Verlag New York, Inc., pp. 103–122.

[4] BEYER, D., HENZINGER, T. A., JHALA, R., AND MAJUMDAR, R. The software model checker Blast: Applications to software engineering. *International Journal of Software Tools Technology Transfer 9*, 5 (2007), 505–525.

[5] BIERE, A., CIMATTI, A., CLARKE, E. M., STRICHMAN, O., AND ZHU, Y. Bounded model checking. *Advances in Computers 58* (2003), 118–149.

[6] CIFUENTES, C. Parfait – a scalable bug checker for C code. Tool demonstration. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation* (2008).

[7] CIFUENTES, C., HOERMANN, C., KEYNES, N., LI, L., LONG, S., MEALY, E., MOUNTENEY, M., AND SCHOLZ, B. Begbunch – benchmarking for C bug detection tools. In *International Workshop on Defects in Large Software Systems, Chicago, Illinois.* (July 2009).

[8] CIFUENTES, C., AND SCHOLZ., B. Parfait – designing a scalable bug checker. In *Proceedings of The ACM SIGPLAN Static Analysis Workshop* (June 2008).

[9] CLARKE, E., BIERE, A., RAIMI, R., AND ZHU, Y. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des. 19*, 1 (2001), 7–34.

[10] CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)* (2004), K. Jensen and A. Podelski, Eds., vol. 2988 of *Lecture Notes in Computer Science*, Springer, pp. 168–176.

[11] CLARKE, JR., E. M., GRUMBERG, O., AND PELED, D. A. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[12] COUSOT, P., AND COUSOT, R. Modular static program analysis. In *Proceedings of Compiler Construction* (2002), Springer-Verlag, pp. 159–178.

[13] DEUTSCH, A. Static verification of dynamic properties. polyspace white paper, February 2004.

[14] ENGLER, D. Static analysis versus model checking for bug finding. 1–1.

[15] ENGLER, D. R., AND MUSUVATHI, M. Static analysis versus software model checking for bug finding. In *VMCAI* (2004), B. Steffen and G. Levi, Eds., vol. 2937 of *Lecture Notes in Computer Science*, Springer, pp. 191–210.

[16] EVANS, D., LAROCHELLE, D., AND ATTACKS, S. Improving security using extensible lightweight static analysis, 2002.

[17] HECKMAN, S., AND WILLIAMS, L. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *ESEM* (2008), pp. 41–50.

[18] JOHNSON., S. Lint, a C program checker. Unix programmers manual, 1978.

[19] KRATKIEWICZ, K. Using a diagnostic corpus of C programs to evaluate buffer overflow detection by static analysis tools. In *In Proc. BUGS05* (2005), p. 05.

[20] LIVSHITS, V. B., AND LAM, M. S. Tracking pointers with path and context sensitivity for bug detection in C programs. In *ESEC / SIGSOFT FSE* (2003), pp. 317–326.

[21] LU, S., LI, Z., QIN, F., TAN, L., ZHOU, P., AND ZHOU, Y. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools* (2005).

[22] LUECKE, G., COYLE, J., HOEKSTRA, J., KRAEVA, M., XU, Y., PARK, M., KLEIMAN, E., WEISS, O., WEHE, A., AND YAHYA, M. The importance of run-time error detection. http://rted.public.iastate.edu/.

[23] MERZ, S. *Model checking: a tutorial overview*. Springer-Verlag New York, Inc., New York, NY, USA, 2001, pp. 3–38.

[24] MIT, M. E., AND E., M. D. Static and dynamic analysis: Synergy and duality. In *In WODA 2003: ICSE Workshop on Dynamic Analysis* (2003), pp. 24–27.

[25] MYERS, G. J., AND SANDLER, C. *The Art of Software Testing*. John Wiley & Sons, 2004.

[26] OF STANDARDS, N. N. I., AND TECHNOLOGY. Samate reference dataset (srd) project., January 2006. http://samate.nist.gov/SRD.

[27] XIE, Y., AND AIKEN, A. Saturn: A SAT-based tool for bug detection. In *of Lecture Notes in Computer Science* (2005), Springer, pp. 139–143.