Exploiting Undefined Behaviors for Efficient Symbolic Execution

Asankhaya Sharma
Department of Computer Science
National University of Singapore
asankhs@comp.nus.edu.sg

ABSTRACT

Symbolic execution is an important and popular technique used in several software engineering tools for test case generation, debugging and program analysis. As such improving the performance of symbolic execution can have huge impact on the effectiveness of such tools. On the other hand, optimizations based on undefined behaviors are an essential part of current C and C++ compilers (like GCC and LLVM). In this paper, we present a technique to systematically introduce undefined behaviors during compilation to speed up the subsequent symbolic execution of the program. We have implemented our technique inside LLVM and tested with an existing symbolic execution engine (Pathgrind). Preliminary results on the SIR repository benchmark are encouraging and show 48% speed up in time and 30% reduction in the number of constraints.

1. MOTIVATION

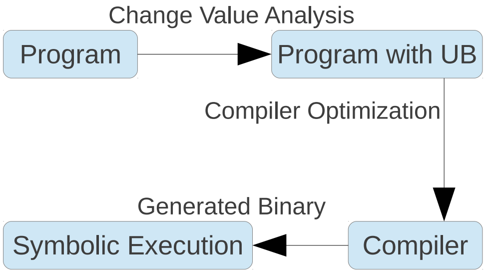
Software engineering tools [6, 9] routinely employ symbolic execution for various applications like automated test case generation [20, 23], bug finding [7, 22], debugging [14, 8], performance analysis [16, 27] and verification [19, 15]. Symbolic execution (when used as a dynamic analysis) is based on direct execution of the program and can easily handle any calls to external libraries or OS by concretizing arguments [25]. This enables such tools to analyze many real-world programs and software. Moreover unlike static analysis, dynamic symbolic execution does not use abstraction which eliminates spurious counter examples and false positives. When coupled with a path exploration based technique like DART [13], dynamic symbolic execution can try to explore all the paths within a program to provide a degree of completeness to the analysis. Due to several practical applications, improving the performance of symbolic execution can have a big impact. Two main sources of bottlenecks for dynamic symbolic execution are path explosion due to unbounded number of paths in the program and the time taken to solve path conditions (or formulas). Recent advances in constraint solving have led to efficient SMT solvers like Z3 [10] that have helped reduce time taken to solve individual path condition formulas. However, we argue that there is a yet another bottleneck for symbolic execution which is the program itself. Programs routinely exhibit undefined behaviors [30] and from the point of view of symbolic execution such behavior can also lead to bugs. C and C++ compilers perform optimizations based on undefined behaviors that enable them to generate code which is faster although sometimes unexpected [29]. Undefined behaviors have been a leading cause of integer overflow bugs [11] in compilers. Are undefined behaviors always a bane or can they be a boon when it comes to speeding up symbolic execution? The research problem we are trying to address in this work is, can optimizations based on undefined behaviors in existing C and C++ compilers be used to speed up symbolic execution.

2. RELATED WORK

As symbolic execution is an active research area there are several important work addressing the scalability and performance issues. Existing symbolic execution engines like KLEE [6], EXE [7] and Klover [20] use techniques based constraint subsumption, simplification and slicing to avoid making unnecessary calls to the SMT solver. Domain-specific and contextual information about the program can be used to optimize the performance of constraint solvers as shown in [12]. In addition, concolic symbolic execution is based on concretizing arguments that cannot be symbolically evaluated. Concrete values also lead to simpler constraints in formulas that are much easier to solve. Moreover, there has been work to address the path explosion program in symbolic execution using efficient state merging [17], multi path analysis [9] and symbolic program decomposition [24]. As programs in general have unbounded number of paths the problem of path explosion in some sense is unavoidable. But by focusing on a subset of paths (for bug finding and debugging) within the program, symbolic execution can be made to preferentially explore relevant portions of program. Caching, reusing and memoizing [31] constraints based on path conditions can improve the overall performance and effectiveness of symbolic execution even across different runs, programs [28] and compilers [26]. Above techniques focus mostly on the size and complexity of formulas generated during path exploration and the number of paths explored during symbolic execution. Another source hampering the scalability of symbolic execution is the program itself. A prior static analysis of the program can be used to compute useful information to speed up subsequent dynamic symbolic execution. The computed information can be used to transform the program based on the relevancy of certain functions for symbolic execution [21] or dependence of modules within the program [5]. However, in these techniques the symbolic execution engine also needs to be modified in order to take advantage of the information computed by the prior static analysis. The combination of undefined behaviors in C and C++ compilers and optimizations based on them to speed up symbolic execution is missing in all these approaches.

3. APPROACH

The uniqueness of our approach is to define a technique that introduces undefined behaviors in the program but does not change the semantics of the program. Then, existing compiler optimizations based on undefined behaviors speed up the symbolic execution. The following figure illustrates the method.



The program under consideration for symbolic execution undergoes a transformation using change value analysis (CVA) [26]. CVA is a static analysis based on the output variables in the program. It computes the set of variables whose values do not change with a change in the value of the output variables. We perform a control and data flow sensitive analysis to compute a fixed point over a lattice of three points denoting the state of a variable, viz. *Changed*, *Unchanged* and *Undefined*. Initially, all the program variables are marked *Undefined*. At the beginning of the analysis the output variables are marked *Changed*, then information about *Changed* variables is propagated backwards using sound transfer functions (computing a fixed point over the lattice) to mark all the dependent variables as *Changed*. In the end we replace all the *Unchanged* and *Undefined* variables with a compiler specific symbol (e.g. `Undef` for LLVM) that takes a non-deterministic value. This gives us a transformed program that can exhibit undefined behaviors (UB) due to the presence of non-deterministic value in some variables. Existing C and C++ compilers exploit the non-deterministic values for optimizations and generate a much smaller and simpler binary. Consider the following code snippet, CVA determines that the value of variable `z` is unchanged and replaces it `Undef`. Later, compiler optimizations eliminate 3 lines of code as shown below.

```
int x, y, z; //input
int a;
a_c=z_uc;
if(x_c−y_c>0)
  a_c=x_c;
else a_c=y_c;
if(z_uc>a_c)
  printf("z is max");
return a_c;
```

(Before CVA)

```
int x, y, z; //input
int a;
if(x−y>0)
  a=x;
else a=y;
return a;
```

(After CVA)

The benefits of tricking the compiler to do transformations based on undefined values are fourfold, firstly it reuses the existing optimizations in compilers for undefined behaviors to do elimination of code that is not relevant to symbolic execution of program, secondly it enables one to use a simpler and faster static analysis (CVA), thirdly it does not require any change in the symbolic execution engine to use the results from the static analysis during dynamic path exploration and finally it allows reduction in the size of the generated binaries for the program even before applying the subsequent constraint solving and path exploration optimizations.

We also note the similarity of our approach to [2] which is effective in pruning redundant executions in compilers by introducing undefined behaviors based on data flow analysis. However, to the best of our knowledge this is the first time, undefined behaviors are used for improving or optimizing a software engineering technique. Empirical studies on dynamic symbolic execution across different programs [28] and different compilers [26] have already shown potential for improving symbolic execution. When compared to solvers that eliminate irrelevant constraints during dynamic symbolic execution, CVA works by removing portions of program that do not affect the output and thus prevents irrelevant constraints from getting generated in the first place.

4. RESULTS

We have implemented the approach as a new compiler pass (CVA) inside LLVM [18]. The source code for entire development is available (under GNU GPL v3) at [1]. In order to evaluate our techniques we used an existing symbolic execution engine Pathgrind [3, 26] and tested it with programs from the SIR repository [4]. We used Pathgrind as it does dynamic symbolic execution from binaries and does not require instrumentation or access to source code. In future we plan to do more experiments with other popular symbolic execution engines (like KLEE, Klover etc.). For the experiments, we built two set of binaries one without CVA and another with CVA (which were on average 14% smaller). Then we used Pathgrind to symbolically execute the two binaries up to a certain fixed depth bound. This ensures that in both the binaries Pathgrind executes the same number of paths symbolically. The following table summarizes the preliminary results from the experiments.

Program	LoC	Constraints		Time	
			(CVA)		(CVA)
tcas	173	848	601	43.7	24.2
schedule2	374	960	821	78.4	34.6
replace	564	264	219	53.9	39.7
totinfo	565	256	210	24.7	11.8
print_tokens2	570	632	632	180.9	78.5
space	6199	100	91	82.6	52.5
grep	10068	512	56	55.3	19.3
flex	10459	576	340	180.5	101
sed	14427	144	17	13.9	7.5
Total	43399	4292	2987	714.02	369.06

For all the programs (except `print_tokens2`) there is reduction in the number of constraints generated during symbolic execution. Moreover, there is also reduction in the overall time taken for symbolic execution of all the programs The `print_tokens2` program does printing and character manipulation of the input string. Thus it does not present much opportunity to exploit unchanged variables. For `grep` and `sed` programs, the first input (regular expression) is kept constant while the second input (file) is changed to generate different test cases. In this case parts of program that correspond to the first input do not affect the output, thus many unchanged variables are eventually eliminated. One threat to the validity of this study is that real-world code may not have the characteristics that enable optimizations with undefined behaviors. We plan to experiment our approach on a wider range of programs from different domains in future. Nevertheless, even in this small set (9) of programs of moderate size (upto 10k Loc) we see a reduction of 48% in time and 30% in number of constraints.

5. CONTRIBUTIONS

The main contribution of this research is to show that systematically introducing undefined behaviors can speed up symbolic execution. As C and C++ compilers get even better at exploiting undefined behaviors this technique will enable corresponding symbolic execution engines to get more efficient as well. Thus undefined behaviors in programs are not always a bad thing and can actually be useful in certain contexts.

6. REFERENCES

[1] Change Value Analysis. http://github.com/codelion/pa.llvm/tree/master/CVA. [Online; accessed 21-Nov-2013].

[2] It's Time to Get Serious About Exploiting Undefined Behavior. http://blog.regehr.org/archives/761. [Online; accessed 21-Nov-2013].

[3] Pathgrind. https://github.com/codelion/pathgrind. [Online; accessed 21-Nov-2013].

[4] Software-artifact Infrastructure Repository. http://sir.unl.edu. [Online; accessed 21-Nov-2013].

[5] Saswat Anand, Alessandro Orso, and Mary Jean Harrold. Type-dependence analysis and program transformation for symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 117–133. Springer, 2007.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224. USENIX Association, 2008.

[7] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[8] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 121–130. IEEE, 2011.

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *ACM SIGARCH Computer Architecture News*, 39(1):265–278, 2011.

[10] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.

[11] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in c/c++. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 760–770, Piscataway, NJ, USA, 2012. IEEE Press.

[12] Ikpeme Erete and Alessandro Orso. Optimizing constraint solving to better support symbolic execution. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 310–315. IEEE Computer Society, 2011.

[13] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[14] Reiner Hähnle, Marcus Baum, Richard Bubel, and Marcel Rothe. A visual interactive debugger based on symbolic execution. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 143–146. ACM, 2010.

[15] Joxan Jaffar, Jorge A Navas, and Andrew E Santosa. Unbounded symbolic execution for program verification. In *Runtime Verification*, pages 396–411. Springer, 2012.

[16] D Kebbal. Automatic flow analysis using symbolic execution and path enumeration. In *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, pages 8–pp. IEEE.

[17] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *ACM SIGPLAN Notices*, 47(6):193–204, 2012.

[18] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[19] Quang Loc Le, Asankhaya Sharma, Florin Craciun, and Wei-Ngan Chin. Towards complete specifications with an error calculus. In *NASA Formal Methods*, 2013.

[20] Guodong Li, Indradeep Ghosh, and Sreeranga P Rajan. Klover: A symbolic execution and automatic test generation tool for c++ programs. In *Computer Aided Verification*, pages 609–615. Springer, 2011.

[21] Xin Li, Daryl Shannon, Indradeep Ghosh, Mizuhito Ogawa, Sreeranga P Rajan, and Sarfraz Khurshid. Context-sensitive relevancy analysis for efficient symbolic execution. In *Programming Languages and Systems*, pages 36–52. Springer, 2008.

[22] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *Static Analysis*, pages 95–111. Springer, 2011.

[23] Mike Papadakis and Nicos Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *Software reliability engineering (ISSRE), 2010 IEEE 21st international symposium on*, pages 121–130. IEEE, 2010.

[24] Raul Santelices and Mary Jean Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 195–206. ACM, 2010.

[25] Asankhaya Sharma. A critical review of dynamic taint analysis and forward symbolic execution. Technical report, 2012.

[26] Asankhaya Sharma. An empirical study of path feasibility queries. *CoRR*, abs/1302.4798, 2013.

[27] Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using ranged analysis. In *ACM SIGPLAN Notices*, volume 47, pages 523–536. ACM, 2012.

[28] Willem Visser, Jaco Geldenhuys, and Matthew B Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 58. ACM, 2012.

[29] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined behavior: What happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pages 9:1–9:7, New York, NY, USA, 2012. ACM.

[30] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 260–275, New York, NY, USA, 2013. ACM.

[31] Guowei Yang, Corina S Păsăreanu, and Sarfraz Khurshid. Memoized symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 144–154. ACM, 2012.