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 [20,24], bug finding [7,22], debugging [13,17], 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 [23]. This enables such tools to analyze many real-world programs and software. Moreover unlike static analysis, dynamic symbolic execution does not use abstractions 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 = z;      //input
if(x > y>0)
a = x;
else a = y;
return a;
```

(Before CVA) (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 [3]. 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>LoC</th>
<th>Constraints</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>173</td>
<td>848</td>
<td>601</td>
</tr>
<tr>
<td>schedule2</td>
<td>374</td>
<td>960</td>
<td>821</td>
</tr>
<tr>
<td>replace</td>
<td>564</td>
<td>264</td>
<td>219</td>
</tr>
<tr>
<td>totinfo</td>
<td>565</td>
<td>256</td>
<td>210</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>570</td>
<td>632</td>
<td>632</td>
</tr>
<tr>
<td>space</td>
<td>6199</td>
<td>100</td>
<td>91</td>
</tr>
<tr>
<td>grep</td>
<td>10068</td>
<td>512</td>
<td>56</td>
</tr>
<tr>
<td>flex</td>
<td>10459</td>
<td>576</td>
<td>340</td>
</tr>
<tr>
<td>sed</td>
<td>14427</td>
<td>144</td>
<td>17</td>
</tr>
<tr>
<td>Total</td>
<td>43399</td>
<td>4292</td>
<td>2987</td>
</tr>
</tbody>
</table>

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


