Targeted Program Transformations for Symbolic Execution

Cristian Cadar
Imperial College London, UK
c.cadar@imperial.ac.uk

ABSTRACT

Semantics-preserving program transformations, such as refactorings and optimisations, can have a significant impact on the effectiveness of symbolic execution testing and analysis. Furthermore, semantics-preserving transformations that increase the performance of native execution can in fact decrease the scalability of symbolic execution.

Similarly, semantics-altering transformations, such as type changes and object size modifications, can often lead to substantial improvements in the testing effectiveness achieved by symbolic execution in the original program.

As a result, we argue that one should treat program transformations as first-class ingredients of scalable symbolic execution, alongside widely-accepted aspects such as search heuristics and constraint solving optimisations. First, we propose to understand the impact of existing program transformations on symbolic execution, to increase scalability and improve experimental design and reproducibility. Second, we argue for the design of testability transformations specifically targeted toward more scalable symbolic execution.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – Symbolic execution

Keywords

Testability transformations, dynamic symbolic execution

1. BACKGROUND

Dynamic symbolic execution (DSE) [3] has gained tremendous popularity in the last decade, becoming part of the standard toolbox of techniques in many computer science fields including software engineering, programming languages, software testing, verification, security, and computer systems. The technique has enabled a wide range of applications, including automatic detection of bugs and security vulnerabilities, recovery of corrupt documents, patch generation, and automatic debugging, among many others.

At a high level, DSE is a program analysis technique that allows the automatic exploration of paths in a program. It works by executing the program on a symbolic input, which is initially unconstrained. As the program runs, any operations that depend on the symbolic input add constraints on the input. For example, if the program input is represented by variable `x`, than the statement `y = x + 1` would add the constraint that $y = x + 1$. Most importantly, whenever a branch that depends (directly or indirectly) on the symbolic input is reached, the technique first checks if both sides of the branch are feasible, and if so, it forks execution and follows each side separately, adding the constraint that the branch condition is true on the *then* side and false on the *else* side. For example, given the symbolic input x, the symbolic execution of the branch `if (x == 10)` would result in two paths being explored, one on which $x = 10$ and one on which $x \neq 10$.

There are two main challenges in DSE: path explosion and constraint solving. In all but the smallest programs, the number of paths is extremely large, being typically at least exponential in the number of static branches in the code. In the presence of loops, path explosion gets even worse, as a loop with n iterations having inside a branch that depends on the symbolic input can spawn up to 2^n paths. Existing research has proposed a range of solutions, such as search heuristics, redundant path elimination, function summaries and path merging. While these solutions have made the approach practical for several types of applications, path explosion still represents an important challenge in DSE.

The second main challenge is related to constraint solving. The problem is compounded by two factors: first, DSE can generate large expensive constraints, and second, DSE issues a constraint solving query *at every single* branch that depends on a symbolic value, to determine the feasibility of each side of the branch. Despite significant progress in constraint solving technology during the recent years [5], constraint solving continues to be one of the main bottlenecks of DSE, and it is absolutely essential for the success of DSE to devise effective techniques that target the kind of constraints generated during symbolic execution.

2. TARGETED PROGRAM TRANSFORMATIONS FOR SYMBOLIC EXECUTION

We believe that the scalability of dynamic symbolic execution can be improved via targeted program transformations, referred in literature as *testability transformations* [8]. The first insight is that semantically-equivalent programs can differ substantially with respect to the effectiveness of

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2803205

```
int get_value(int k) {
    return k*k*k;
}

// precond: 0 <= k < 1000
int foo(unsigned k) {
  if (get_value(k)    > 100000 ||
      get_value(k−1) > 100000)
      return 0;
  else return 1;
}
```

(a)

```
int values[1000] = {0, 1, 8, 27, 64, 125,
216, 343, 512, 729, 1000, 1331, 1728, 2197,
2744, 3375, 4096, 4913, 5832, ...};

// precond: 0 <= k < 1000
int foo(unsigned k) {
  if (values[k]    > 100000 ||
      values[k−1] > 100000)
      return 0;
  else return 1;
}
```

(b)

Figure 1: The C code in (a) is transformed into the code in (b) using a precomputed lookup table.

DSE to explore the program state space. These differences can be significant: as we are going to show below, simple semantics-preserving transformations can lead to orders of magnitude difference in performance. The second insight is that semantics-altering transformations can also substantially improve the performance of symbolic execution and the quality of testing of the *original* program.

2.1 Semantics-Preserving Transformations

One might be initially surprised to find out that the scalability of symbolic execution testing can vary dramatically across *semantically-equivalent* programs. Furthermore, it can be even more unexpected that transformations that *increase* the performance of native execution can in fact *decrease* the scalability of symbolic execution. However, this is exactly what often happens in practice, requiring us to rethink the type of transformations that should be applied in a DSE context, and to consider the opportunity of designing transformations that are targeted toward more scalable symbolic execution.

Program transformations can have an impact on both constraint solving and path exploration. Below, we give a couple of examples in each category.

Transformations affecting constraint solving. To illustrate the potential impact of program transformations on constraint solving, consider the function `foo()` in Figure 1a, which we assume is called with an argument between 0 (inclusive) and 1000 (exclusive). In turn, `foo()` calls `get_value()` twice. If `foo()` is called many different times, then precomputing the result of `get_value()` can lead to significant performance gains. Assume the developer hand-optimised the function into the code of Figure 1b. While this speeds up native execution, the performance of DSE drops significantly. For example, running *foo* a single time with the symbolic executor KLEE[1] by treating its integer argument as symbolic takes 0.2 seconds[2] for the code in Figure 1a, and 50 seconds for the code in Figure 1b, which represents a 250 times slowdown! The reason is that without the optimisation, the constraint solver has to deal with satisfiability queries of the form $k * k * k > 100000$, while with the optimisation, it has to deal with queries of the form $(values[k] > 100000) \wedge (values[0] = 0) \wedge (values[1] = 1) \wedge ...$, the latter creating a much larger formula in the solver. In our experience, DSE often struggles with large tables of constants, and one way to deal with this problem would be to reverse such precomputed lookup table optimisations. In other words, a DSE-friendly testability transformation for code fragments involving large constant tables would be to replace them with mathematical formulas. For example, a simple polynomial interpolation method [7] might often be successful.

As a second example, consider common optimisations performed by modern compilers, such as constant folding, dead store elimination, inline expansion, loop fission, loop-invariant code motion, loop unrolling, and test reordering. While many compiler optimisations do help symbolic execution—and our tool KLEE implements various compiler-style optimisations in order to improve performance [1]—surprisingly, some compiler optimisations that significantly improve native program performance can in fact *significantly hurt* the symbolic execution of a program [6].

For example, the code in Figure 2a is optimised via strength reduction into the code in Figure 2b. The strength reduction optimisation aims to replace expensive operations such as multiplication with less expensive ones such as addition. In this example, the multiplication of the loop index is replaced with addition by using the auxiliary variable k. While this optimisation does result in faster native code, the performance of DSE degrades significantly. Treating variable c as the symbolic input, the code in Figure 2a takes 2.2 seconds to run with KLEE, while the code in Figure 2b takes 20 minutes, which represents a 545 times slowdown! The reason is that the latter constructs very large chains of addition operations, which are much more expensive for the underlying constraint solver than multiplications by a constant. Therefore, a DSE-friendly testability transformation would be the reverse of strength reduction—called induction variable substitution and typically applied in order to parallelise loops.

Transformations affecting path exploration. Program transformations could also have a significant effect on path exploration. As a first example, consider the code in Figure 3, which involves a `switch` statement and which has a division by zero bug for `x≠1,2,3,4` and `y=0`. A `switch` statement is a higher-level programming language construct, and most DSE tools would first transform it into lower-level constructs (or allow an external compiler to do so). The way in which the `switch` statement is transformed has a significant influ-

[1] KLEE [1] is a state-of-the-art dynamic symbolic execution engine based on the LLVM compiler infrastructure. It is available as open-source at `http://klee.github.io/`

[2] On an Intel(R) Core(TM)2 Duo CPU E8400 at 3.00GHz, with 8GB RAM, using KLEE based on LLVM 2.9.

```
int foo(int c) {
  int y[500], i;

  for (i = 0; i < 500; i++) {
      y[i] = c * i;

      if (y[i] % 2 == 0)
         printf("Yes\n");
      else printf("No\n");
   }
}
```

(a)

```
int foo(int c) {
  int y[500], i;
  int k = 0;
  for (i = 0; i < 500; i++) {
      y[i] = k; k = k + c;

      if (y[i] % 2 == 0)
         printf("Yes\n");
      else printf("No\n");
   }
}
```

(b)

Figure 2: The C code in (a) is transformed via strength reduction into the code in (b). Conversely, the code in (b) is transformed via induction variable substitution into the code in (a).

ence on the path exploration and the ability to find the bug. For instance, if the transformation employs a binary search algorithm on the switch expression range, KLEE configured to use breadth-first search (BFS) takes 2.3 seconds to hit the division by zero bug. By contrast, if the transformation creates a linear chain of `if` statements, then KLEE does not find the bug within a time limit of one hour. While this result is unsurprising if we think about the structure of the resulting control-flow graph, it shows that program transformations can have an important impact on path exploration in DSE, and symbolic execution tools could use them to their advantage. For example, if a broad exploration of the search space is desired, then one should use BFS and transform `switch` statements using binary search.

As a second simple example, consider the code in Figure 4, which counts the number of positive numbers in an array of ten elements, and reports `Success` if all of them are positive. Compiling this program with LLVM using optimisation flag `-O0` results in 1024 paths explored by KLEE, taking a total of 23 seconds. However, compiling this program with LLVM using optimisation flag `-O2` results in only two paths being explored in 0.04 seconds. The reason is that the `-O2` optimisations transform the branch inside the loop into a `select` operation (`count = select(a[i] > 0, count+1, count)`), which KLEE sends directly to the constraint solver without having to fork. In essence, the optimisation has merged the paths inside the loop!

Transformations involving `switch` and `select` statements are not the only examples: other program transformations such as those splitting or merging loops are likely to have a similar effect on path exploration, and should be treated as another mechanism to improve program exploration, similarly to how search heuristics are designed.

As a final example applicable to both constraint solving and path exploration, consider running DSE with the same settings (e.g., search heuristics) on different versions of the same program: the original source code, the code compiled to an intermediate language, the x86 binary, the source code raised from the binary, etc. While all these programs are semantically equivalent, the performance of DSE can vary significantly; thinking about these versions in terms of semantics-preserving program transformations might provide a better understanding of the tradeoffs involved in operating at different levels of abstraction.

```
int expensive(int x) {
  int bits = 0, i;
  for (i=0; i<32; i++)
      if (x & (1 << i))
         bits++;
  return bits;
}

int foo(int x, int y) {
  switch (x) {
     case 1: return expensive(y+1);
     case 2: return expensive(y+2);
     case 3: return expensive(y+3);
     case 4: return expensive(y+4);
     default: return x/y;
  }
}
```

Figure 3: Example showing the impact of `switch` transformations on DSE.

2.2 Semantics-Altering Transformations

As argued in prior work [8], program transformations that do not preserve the semantics of the program can nevertheless improve the testing of the original program. In the context of symbolic execution, this can be an effective mechanism for improving its scalability.

One example are transformations whose main goal is to reduce the scope of the analysis to a subset of the allowable program behaviours. As a first example, consider a program that uses floating-point numbers. Most DSE engines cannot handle symbolic floating-point computations, mainly due to the poor scalability of floating-point constraint solvers [4]. A possible testability transformation would be to replace all floating-point variables with integers or rational numbers, allowing DSE to make progress, while exploring a limited subset of behaviours. Of course, testing a subset of program behaviours is better than not being able to test a program at all, and in practice it may often be enough to discover important errors or cover interesting parts of the code. Note also that such a transformation would not strictly explore a subset of possible behaviours; due to differences in the arithmetic overflow behaviour between floating point numbers and integers/rationals, some infeasible executions might be

```
int foo(int a[10]) {
  int count=0, i;
  for (i=0; i<10; i++)
      if (a[i] > 0)
         count++;

  if (count == 10)
     printf("Success\n");

  return count;
}
```

Figure 4: Example showing the impact of compiler optimisations of path exploration.

introduced. But this might be easily tolerable in practice, especially since DSE has the ability to generate concrete test inputs, which for example could be rerun to confirm any reported bugs.

As a second example, one could automatically shrink large buffers that pose scalability challenges in DSE, or assign concrete values to parts of a symbolic input. Such testability transformations are often performed manually by DSE testers, but could be easily automated.

3. ENVISAGED IMPACT

Targeted program transformations can have a significant impact on symbolic execution. First, they can lead to increased scalability: while the examples in this paper are toy programs, they illustrate well the way in which testability transformations could help address the two main challenges of DSE, namely speed up constraint solving and improve path exploration.

Second, they can improve experimental design and reproducibility of results, as program transformations are often implicitly performed by the compiler or the analysis framework. This potential confounding factor is almost always overlooked during experimental design, rendering comparisons across projects difficult to assess. For example, starting with the same source code and using the same algorithmic settings, a tool running on top of CIL[3] (such as EXE [2]) and one running on top of LLVM[4] (such as KLEE [1]) can perform very differently, simply because CIL and LLVM apply different transformations to the original program code. Even more, the same tool with the same settings, but using two different versions of the underlying compiler—say KLEE on top of LLVM 2.3 and KLEE on top of LLVM 2.9—can result in widely different behaviour, simply because the different compiler versions perform a (slightly) different set of optimisations!

4. RELATED WORK

This paper is inspired by the author's experience with the significant variations in KLEE's performance when changing LLVM versions or refactoring the code under analysis; this has also been reported by other KLEE users. Recent work by Wagner et al. [9] and Dong et al. [6] more rigorously showed that compiler optimisations can influence DSE performance. In particular, Wagner et al. argued for, and designed, a series of compiler optimisations focusing on improving path exploration in DSE. In this paper, we provide additional evidence of the effect of semantics-preserving trasformations (including compiler optimisations) on both path exploration and constraint solving, and go beyond semantics-preserving optimisations to also look at semantics-altering ones.

The idea of using program transformations to improve testability was first introduced by Harman et al. [8] in the context of search-based software testing. For example, in evolutionary testing, boolean flags involved in branch predicates induce two plateaux on the search space, making evolutionary testing no better than random testing; an effective solution to this problem is to perform a program transformation that replaces the flag variable with the expression that was used to compute it.

5. CONCLUSION

This paper argues for treating program transformations as first-class ingredients of symbolic execution, alongside widely-accepted aspects such as constraint solving and search heuristics. Program transformations occur in many different forms—e.g., refactorings, manual optimisations, translation into intermediate languages and compiler optimisations—and can have a significant impact on the performance of DSE. Understanding existing optimisations can allow developers to disable the ones that hurt scalability and enable the ones that increase it, as well as to improve experimental design and reproducibility. Furthermore, we believe that the design of DSE-friendly testability transformations can have the potential of becoming an important ingredient of scalable symbolic execution testing and analysis.

Acknowledgements. I thank Mark Harman and Paul Marinescu for their feedback, and the EPSRC for supporting this research through an EPSRC Early-Career Fellowship.

6. REFERENCES

[1] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*.

[2] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *CCS'06*.

[3] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *CACM*, 56(2):82–90, 2013.

[4] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *EuroSys'11*.

[5] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *CACM*, 54(9):69–77, Sept. 2011.

[6] S. Dong. An empirical study of the influence of compiler optimizations on symbolic execution. Master's thesis, University of Texas at Austin, 2014.

[7] R. W. Hamming. *Numerical Methods for Scientists and Engineers.* Dover Publications, 1987.

[8] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *TSE*, 30(1):3–16, Jan. 2004.

[9] J. Wagner, V. Kuznetsov, and G. Candea. -Overify: Optimizing programs for fast verification. In *HotOS'13*.

[3] `https://www.cs.berkeley.edu/~necula/cil/`

[4] `http://llvm.org/`