Shadow Symbolic Execution for Better Testing of Evolving Software[*]

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

Hristina Palikareva
Department of Computing
Imperial College London, UK
h.palikareva@imperial.ac.uk

ABSTRACT

In this idea paper, we propose a novel way for improving the testing of program changes via symbolic execution. At a high-level, our technique runs two different program versions in the same symbolic execution instance, with the old version effectively *shadowing* the new one. In this way, the technique can exploit precise dynamic value information to effectively drive execution toward the behaviour that has changed from one version to the next. We discuss the main challenges and opportunities of this approach in terms of pruning and prioritising path exploration, mapping elements across versions, and sharing common symbolic state between versions.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Reliability;
D.2.5 [**Testing and Debugging**]: Symbolic execution

General Terms

Reliability, Verification

Keywords

Patch testing, longitudinal program analysis

1. INTRODUCTION

Software evolves on a continuous basis—developers fix bugs, improve existing code, and implement new features. Unfortunately, often many of these changes themselves give rise to a significant amount of program bugs: new code is by definition little tested in the field, and often introduces bugs that affect both old and new functionality [10,18]. Even relatively simple, self-contained patches which are meant to fix specific bugs often introduce new errors [10,28].

Software testing and analysis have seen tremendous progress in the last few years, with many open-source and commercial bug-finding tools readily available [5,6,8,26]. While these tools generally focus on whole-program testing, recent work has started to look more closely at incremental or longitudinal analysis [1–3,9,15–25,27] where the effort is focused on recently changed code.

One of the techniques which has already shown promise in the context of longitudinal testing is symbolic execution [4, 14], a technique which can systematically explore paths in a program. For example, research on different flavours of *directed symbolic execution* [1–3, 16–18, 21, 22, 24, 27] has considered the problem of generating program inputs that execute recently modified code ("the patch").

At a high-level, these techniques symbolically run only the new program version, taking into account information obtained statically by analysing the differences between the old and the new version. For example, KATCH [18] uses a simple textual differencing algorithm to find out what statements have been added or changed, while DiSE [22] employs static program slicing to determine the statements affected by the patch. While this can lead to significant savings, static analysis of the program differences is often imprecise, and can miss important pruning and prioritisation opportunities, particularly those which exploit dynamic value information. Furthermore, most of the techniques proposed so far have focused on the problem of reaching the patch; while this is key to the success of longitudinal testing, deciding what paths and what inputs to check once the patch is reached is equally challenging and can have an important influence on the quality of patch testing [2,22].

As we show in this paper, precise runtime information about the execution of *both* versions could provide important opportunities for pruning large parts of the program space and for identifying divergent behaviour, potentially improving significantly the scalability of existing longitudinal symbolic execution approaches. To obtain such runtime information, we execute both versions *in the same symbolic execution instance*, with the old version effectively "shadowing" the new one. Therefore, the key research question addressed in this paper is the following:

Can we run two different program versions in the same symbolic execution instance in order to effectively drive execution toward the behaviour that has changed from one version to the other?

In the remainder of the paper, we elaborate on this idea, and discuss the main challenges and opportunities of this approach in terms of pruning and prioritising path exploration,
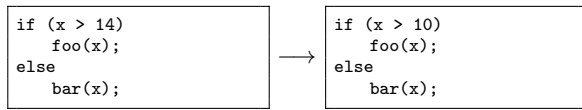
```
if (x > 14)
    foo(x);
else
    bar(x);
```
$\longrightarrow$
```
if (x > 10)
    foo(x);
else
    bar(x);
```

Figure 1: Simple example. The `if` statement is the only change between the two versions.

```
if (x > 0) { ... }
...
if (x > 10)
    assert(false);
```
$\longrightarrow$
```
if (x > 6) { ... }
...
if (x > 8)
    assert(false);
```

Figure 2: Two changes in branch conditions.

mapping elements across versions, and sharing common symbolic state between versions.

2. SHADOW SYMBOLIC EXECUTION

If one changes the branch `if (x > 14)` to `if (x > 10)` in a program as in Figure 1, which values should we pick for the input `x` in order to effectively test the change? Values `x = 0` and `x = 20` may seem appropriate as they exercise both sides of the branch. However, on a closer inspection, these values are useless for the purpose of testing the patch as they would make both program versions behave identically, taking the `then` side of the branch when `x = 20` and the `else` side when `x = 0`. Instead, one should pick a value from the set `{11, 12, 13, 14}`, as these are the values that actually exercise the change, making the old version take the `else` side of the branch, and the new version the opposite `then` side.

An effective analysis would determine that when testing this patch, one does not have to run the `else` side of the branch at all, because for all values of $x \leq 10$ the new version executes identically to the old version. This completely prunes the call to function `bar` and the subsequent analysis of the paths in `bar`, which may bring significant savings, especially if `bar` performs an expensive computation encountering multiple branch conditions dependent on the input. Furthermore, when testing the `then` side of the branch, only values `11, 12, 13` and `14` need to be considered, because the other values do not introduce any changes in the behaviour of the two versions. Drawing these conclusions using only a static analysis of program differences is in general not possible, due to the inherent imprecision of static analysis.

Instead, we propose the concept of *shadow symbolic execution*, in which the two versions of the program are run in the same symbolic execution instance, with the old version effectively *shadowing* the new one. Running the old version in the background can effectively play the role of an oracle that informs the analysis as to what paths and input values need to be tracked in order to effectively test the changes introduced in the new version.

We see several benefits arising from such a form of longitudinal symbolic execution:

(1) A significant reduction in the number of explored paths, resulting from the ability to determine that the two versions would behave identically on some paths.

(2) The ability to prioritise divergences between versions by determining the exact condition/values on which the execution of the two versions diverges. Restricting execution on some paths to a reduced range of values can further prune paths which are incompatible with the reduced value range.

(3) The generation of simpler constraints (and thus faster constraint solving, which is often the bottleneck), by restricting execution on some paths to a reduced value range.

Dynamic symbolic execution (DSE). Before discussing shadow symbolic execution in more detail, we first provide some background information on DSE. In DSE, program inputs are treated as symbolic rather than concrete values,
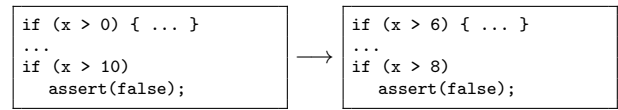
and classes of program paths with the same branching behaviour are encoded as sets of constraints over those symbolic variables. At any point on a path, the symbolic state maintains the current program location, a symbolic store mapping program variables to expressions computed over the symbolic input, and a path condition characterising the inputs that exercise the path. The path condition takes the form of a conjunction of constraints obtained from the symbolic branch conditions encountered along the path. For example, when symbolic execution reaches the `if` statement in the new version in Figure 1, it first checks whether the current path condition allows execution to proceed on both sides of the branch, and if so, it forks execution to follow both paths: on the `then` side it adds the constraint `x > 10` to the path condition, while on the `else` side the constraint `x ≤ 10`.

Multiple changes and four-way forking. In the simple example in Figure 1 where a single conditional `if` statement was modified, we were able to completely cut off the `else` side of the branch along which the two versions behaved identically. In a nutshell, this was because we only considered the *symmetric difference* of the behaviours of the two versions. In practice, the situation becomes more complicated when the patch modifies multiple locations, e.g. two or more branch conditions that make use of a certain symbolic variable. Sometimes even a single change in the code poses extra challenges, e.g. a modified assignment to a variable followed by several branch conditions that make use of that variable. In those cases, the approach of completely pruning branches might become overly aggressive and result in bugs being missed, as we demonstrate below.

Figure 2 illustrates a scenario where the new version changes two symbolic branch conditions. Let us first notice that the new version introduces an assertion failure for two extra values of `x`, namely 9 and 10, and hence contains a regression. Applying the line of reasoning from before, however, we would deduce that the `then` side of the first `if` statement does not introduce any new behaviour (`x > 0` subsumes `x > 6`) and we would discard it as redundant, while on the `else` side of the branch we would narrow the set of interesting values for `x` to the ones in the interval `(0,6]`. While this correctly tests the new behaviour introduced by the change in the first branch, pruning the `then` side at this point would prevent us from testing the change in the second branch, where the regression failure would have occurred.

To address these challenges, we propose a more agile framework that explores the entire branch product but prioritises branches exposing differing behaviour. Figure 3 illustrates the general case: whenever we reach a modified branch condition (modified either directly or via a dependency), instead of forking execution into two paths—one adding the condition `new` and the other `¬new` (as in standard DSE), or one adding the condition `new ∧ ¬old` and the other `¬new∧old` (as when we adopted the symmetric-difference approach)—we fork into four. While leaving the new version on the foreground and adhering to its control flow, we let both its `then` and `else` sides further divide the input space into two: the
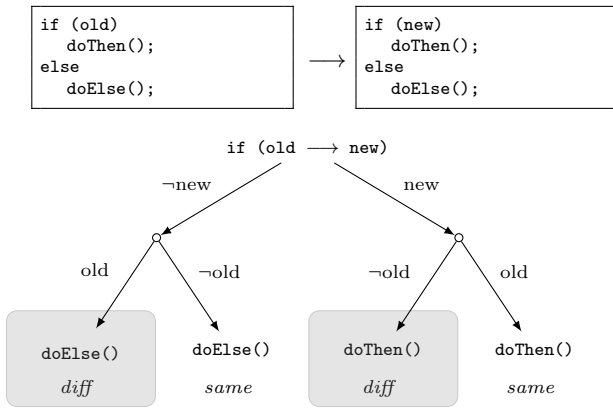
433

```
if (old)          if (new)
   doThen();          doThen();
else              else
   doElse();          doElse();
```

Figure 3: Four-way fork: prioritising branches diverging in control flow (the ones shaded in grey).

```
x = x + 1; y = y + 2;      x = x + 2; y = y + 1;
if (x + y > 10) { ... }    if (x + y > 10) { ... }
else { ... }               else { ... }
```

Figure 4: An opportunity to detect a refactoring.

```
if (x == 0)               if (*p == 0)
   if (y == 0)               { ... }
      { ... }
```

Figure 5: Example of challenges in static matching.

same subbranches model those paths where the new version mimics the old version (e.g. both versions follow the `then` side), while the *diff* subbranches model those paths where the new version diverges (e.g. the new version follows the `then` side, while the old version the `else` side). We note that in practice it is rarely the case that all four subbranches are feasible. In fact, in certain cases we are able to determine, during symbolic execution, that syntactic changes do not introduce a semantic difference (i.e. they are a refactoring), and in this case both *diff* subbranches become infeasible. A contrived illustrative example is depicted in Figure 4.

While the four-way fork strategy might lead in the worst case to an exponential increase in the number of paths to explore, it allows us to effectively prioritise divergent behaviour. Our search would first schedule the cases in which the two versions diverge at this branch point, after which it will explore the cases in which the two versions behave identically, to test subsequent changes.

Finally, note that conditionals are the only points in the program where divergences manifest, although other program statements, in particular assignments, need to be processed so that the symbolic stores (mapping program variables to expressions over the symbolic input) of both versions are maintained throughout execution.

We remark that in certain cases it is possible to determine statically whether symmetric difference suffices or whether a four-way fork is necessary. For example, using static analysis of the control and data dependencies of the program, we can compute the forward slices of the changes and treat the ones with non-intersecting forward slices as independent.

In general, the patch type—e.g. whether its modified locations are interacting [3], or whether its effects are localised to a certain part of the code—can have a big impact on shadow symbolic execution, and one of the aspects that needs to be evaluated is the prevalence in practice of various patch types.

Mapping program elements across versions. So far, we have assumed that the analysis can easily determine what and how parts of the code have changed. However, mapping program elements across versions [13]—undecidable in the general case—can have a big influence on our technique, both in terms of effectiveness and memory consumption.

For ease of exposition, let us first focus on patches that change the code of a single method. There are three types of changes: construct modifications (e.g. changing `if (x > 1)`
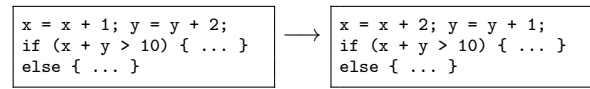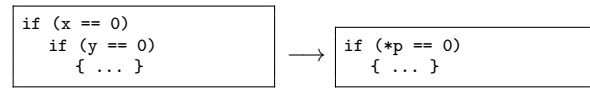
to `if (x > 2)`), additions (e.g. adding an extra assignment or `if` statement), and deletions. Without loss of generality, we propose to model all changes as modifications of existing constructs by adding *dummy* statements at appropriate points in the program [23]. For example, if the new version adds the assignment `x = 4`, then we can add a corresponding dummy statement `x = x` in the old version. Similarly, if the new version removes the check `if (x > 4)`, we can add the dummy check `if (true)` in the new version. The process can be further generalised to the interprocedural case by inlining functions, and unrolling recursive functions up to a certain depth.

The algorithm described above can be implemented statically or dynamically. While the static approach is likely simpler to implement, performing the matching dynamically can once again provide better results (i.e. more path pruning opportunities). As an illustrative example, consider the case in which we try to map the two versions shown in Figure 5. While it may be infeasible to determine statically whether `p` points to `x` or `y`, this information is readily available at runtime. The decision can have an important impact in practice. Suppose that `p` points to `x`, but the algorithm incorrectly matches `if (y == 0)` to `if (*p == 0)`. Then, we would incorrectly decide that at the first branch point there is a change in behaviour (and prune/prioritise accordingly), while in fact the two versions behave identically here.

Reaching the patch. While our approach is primarily concerned with what paths and inputs to check once the patch is reached, patch reachability—undecidable in the general case—is likely to remain a significant challenge. One option is to bypass this problem altogether, by focusing on testing program components (e.g. individual functions) in isolation [22]. For system-level testing, however, patch reachability is essentially a prerequisite for shadow symbolic execution, and existing techniques (based on a combination of search heuristics and static analysis) are a good starting point. However, more precise runtime information might help here too. For example, implementing a weakest precondition analysis that takes into account the values with which we would like to reach a certain branch condition (e.g. `x ∈ {11, 12, 13, 14}` for the patch in Figure 1) is likely to lead to additional paths being pruned.

Efficiently sharing state. As in other instances when different software variants or versions are run together [7, 11, 12, 19, 25], shadow symbolic execution can substantially increase memory consumption. As a result, it is important to maximise sharing between the symbolic states, and in particular the symbolic stores, of the two versions. Since the patch typically affects a relatively small number of memory locations, everything else can be easily shared. Furthermore, it is possible to share those parts of symbolic expressions that are identical between versions.

434

3. RELATED WORK

As discussed in the introduction, most prior work on longitudinal symbolic execution [1, 3, 16–18, 22, 24, 27] used search heuristics and static analysis to identify and exercise those statements that are affected by the patch. Shadow symbolic execution aims to advance the state of the art by focusing on exactly those paths and input values that trigger a different behaviour in the two versions.

Differential symbolic execution [21] is a general framework that can reason about program differences, but its reliance on summaries raises significant scalability issues. Partition-based verification (PRV) [2] uses random testing and concolic execution to infer *differential partitions*, i.e. input partitions that propagate the same differential state to the output. PRV separately runs both program versions using concolic execution, and uses static and dynamic slicing to infer differential partitions. In contrast to PRV, by running the two versions in a synchronised fashion, shadow symbolic execution does not need to re-execute potentially expensive path prefixes and can provide opportunities to prune and prioritise paths early in the execution, as well as to simplify constraints. However, it does not take into account the effect of the differential state on the output.

Overall, while shadow symbolic execution certainly offers new trade-offs and opportunities, it is unlikely to subsume any of the techniques cited above. Testing evolving software is a difficult undecidable problem, which is unlikely to be tamed by any single technique.

The mechanism through which shadow symbolic execution determines the input ranges that trigger a different behaviour in the two versions is to run both program versions in the same symbolic execution instance. Running multiple versions in parallel has been employed in several other contexts, including online validation [19, 25], model checking [7], product line testing [12], and software updating [11]. While some of the mechanisms developed in these contexts will likely prove useful here, longitudinal symbolic execution introduces specific challenges and opportunities in terms of path exploration, state sharing and constraint solving.

4. CONCLUSION

We have argued that shadow symbolic execution—where the two versions of the program are run in the same symbolic execution instance, with the old version *shadowing* the new one—could significantly improve patch testing, by focusing on exactly those paths and input values that trigger the behavioural differences introduced by a software patch. While this approach provides important opportunities, it comes with equally significant challenges, which are waiting to be explored in future work: effectively prioritising behavioural differences for various types of patches, enhancing static analysis with runtime value information, efficiently sharing state between the two versions and effectively mapping program elements across versions, as well as exploring applications of this technique to other related domains such as test suite augmentation, fault localisation, code summarisation, refactoring, and automatic patch generation.

5. REFERENCES

[1] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *ISSTA'11*.

[2] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Partition-based regression verification. In *ICSE'13*.

[3] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Regression tests to expose change interaction errors. In *ESEC/FSE'13*.

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

[5] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS'04*.

[6] Coverity software. http://www.coverity.com.

[7] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. In *ISSTA'07*.

[8] Fortify software. http://www.fortify.com.

[9] B. Godlin and O. Strichman. Regression verification. In *DAC'09*.

[10] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *ICSE'10*.

[11] P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *ICSE'13*.

[12] C. H. P. Kim, S. Khurshid, and D. Batory. Shared execution for efficiently testing product lines. In *ISSRE'12*.

[13] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *MSR'06*.

[14] J. C. King. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.

[15] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *ESEC/FSE'13*.

[16] K.-K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks. Directed symbolic execution. In *SAS'11*.

[17] P. D. Marinescu and C. Cadar. High-coverage symbolic patch testing. In *SPIN'12*.

[18] P. D. Marinescu and C. Cadar. KATCH: High-coverage testing of software patches. In *ESEC/FSE'13*.

[19] M. Maurer and D. Brumley. TACHYON: Tandem execution for efficient live patch testing. In *USENIX Security'12*.

[20] D. Notkin. Longitudinal program analysis. In *PASTE'02*.

[21] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *FSE'08*.

[22] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI'11*.

[23] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *ASE'08*.

[24] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. eXpress: guided path exploration for efficient regression test generation. In *ISSTA'11*.

[25] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *ASPLOS'09*.

[26] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA'04*.

[27] Z. Xu and G. Rothermel. Directed test suite augmentation. In *ASPEC'09*.

[28] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *ESEC/FSE'11*.