Shadow of a Doubt: Testing for Divergences Between Software Versions

Hristina Palikareva* Tomasz Kuchta* Cristian Cadar

Department of Computing
Imperial College London, UK
{h.palikareva, t.kuchta, c.cadar}@imperial.ac.uk

ABSTRACT

While developers are aware of the importance of comprehensively testing patches, the large effort involved in coming up with relevant test cases means that such testing rarely happens in practice. Furthermore, even when test cases are written to cover the patch, they often exercise the same behaviour in the old and the new version of the code.

In this paper, we present a symbolic execution-based technique that is designed to generate test inputs that cover the new program behaviours introduced by a patch. The technique works by executing both the old and the new version in the same symbolic execution instance, with the old version *shadowing* the new one. During this combined shadow execution, whenever a branch point is reached where the old and the new version diverge, we generate a test case exercising the divergence and comprehensively test the new behaviours of the new version.

We evaluate our technique on the `Coreutils` patches from the `CoREBench` suite of regression bugs, and show that it is able to generate test inputs that exercise newly added behaviours and expose some of the regression bugs.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Symbolic patch testing, regression bugs, cross-version checks

1. INTRODUCTION

The malleability of software is both a blessing and a curse. On the one hand, one can easily change software to fix incorrect behaviour or add new functionality. On the other hand, software changes are often responsible for introducing errors and security vulnerabilities, making users think twice about whether or not to update to the latest version.

Ideally, software changes, typically referred to as *patches*, should be comprehensively tested. At the very minimum, each line of code affected by the patch should be covered by at least one test case. While this level of testing is still far from being achieved in practice [21], automatic techniques for enabling high-coverage patch testing are becoming more and more successful [1, 17, 20, 29, 31]. Many of these techniques are based on dynamic symbolic execution [7], a program analysis technique that provides the ability to generate inputs that form high-coverage test suites.

However, achieving full statement or even full branch coverage for the patch code is clearly insufficient. In fact, one can achieve full statement and branch coverage without testing *at all* the new behaviour introduced by the patch! To give a simple example, consider a patch that only changes the statement `if (x > 10)` to `if (x > 20)`, with this statement executed only once by a deterministic program. Suppose that the developer adds two test cases, `x = 0` and `x = 30`, to test the patch. A superficial reasoning might conclude that the change is comprehensively tested, as we have inputs covering each side of the branch. However, the execution of these inputs is completely unaffected by the patch, as the program will behave identically for these inputs before and after the patch is applied. Careful analysis shows that the program behaviour is changed only when `x` is between 11 and 20 (inclusive)—causing the two versions to take different sides of the branch—so one of these values should be used to test the patch.

In this paper, we present a technique based on dynamic symbolic execution that can generate test inputs that cover the new program behaviours introduced by a patch. The technique works by executing both the old (unpatched) version and new (patched) version in the same symbolic execution instance, with the old version shadowing the new one. This form of analysis, which we refer to as *shadow symbolic execution*, makes it possible to (1) precisely determine when the two versions exhibit divergent behaviour, and (2) keep execution time and memory consumption low. Both of these features are key for effective testing of software patches, and are difficult to achieve without running both versions in the same symbolic execution instance.

The main contributions of this paper are:

(1) Shadow symbolic execution, a technique for generating inputs that trigger new behaviours introduced by a patch. The technique effectively prunes a large number of irrelevant execution paths and reduces the program search space.

*The first two authors contributed equally to this paper.

ICSE '16, May 14 - 22, 2016, Austin, TX, USA

ISBN 978-1-4503-3900-1/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2884781.2884845

```
1  char arr[4];
2
3  int foo(int x) {
4    int y = x - 1;
5    if (y > 7) {
6      int z = x - 8;
7      if (z < 4)
8        arr[z] = 'A';
9      return 0;
10   }
11   return 1;
12 }
```

Figure 1: A toy example illustrating symbolic execution.

(2) A way to unify the two program versions and represent them as a single annotated program, equivalent to executing both versions in lockstep, which lets us run the analysis in a single symbolic execution instance. The unified program could be useful in other dynamic analysis techniques.

(3) A tool called SHADOW that implements shadow symbolic execution, and the experience of applying it to the `Coreutils` patches in `CoREBench`, a collection of highly-complex real-world patches.

The rest of the paper is organised as follows. We introduce shadow symbolic execution in §2 and then present it in detail in §3. We then give a brief overview of our prototype tool SHADOW in §4 and describe our experience applying it to test a suite of complex patches in §5. We finally discuss related work in §6 and conclude in §7.

2. OVERVIEW

In our approach, we assume that we already have a test input that *touches* the patch, i.e. executes at least one patch statement—if such an input does not exist in the program's test suite, it could be generated using previous techniques such as KATCH [20].

Given such an input, our technique is designed to automatically generate new inputs that exercise the *new* behaviours added by the patch. These inputs can then be analysed by developers to either uncover bugs (if the new behaviour is unexpected) or create test cases that witness and validate the new behaviour (if it is expected).

Our technique is based on dynamic symbolic execution [7], a popular program analysis technique that runs the program on *symbolic* rather than concrete inputs, with classes of program paths with the same branching behaviour being encoded as sets of constraints over those symbolic inputs. 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 (reflecting dynamic runtime information for the non-symbolic inputs), and a *path condition* (PC) characterising the inputs that exercise the path. The PC takes the form of a conjunction of constraints obtained from the symbolic branch conditions encountered along the path.

As an example, consider the toy program in Figure 1, and assume that we want to run function `foo` on symbolic input `x`. When symbolic execution starts, the symbolic store is $\{\mathtt{x} \to x\}$, meaning that variable `x` maps to symbolic input x, and the PC is `true`. After line 4 is executed, the symbolic store becomes $\{\mathtt{x} \to x,\ \mathtt{y} \to x-1\}$. When execution reaches branch `y > 7`, we discover that under the current PC both

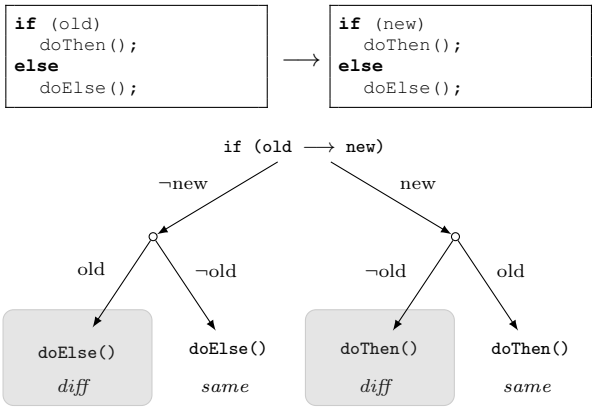

Figure 2: Four-way forking in shadow symbolic execution to capture divergent executions (the ones shaded in grey).

sides of the branch are feasible, so we fork execution, following each path separately. On the `then` branch we add to the PC the constraint $x - 1 > 7$, while on the `else` branch its negation $x - 1 \leq 7$. The latter path immediately terminates by executing `return 1`, but the former continues by executing the assignment `z = x - 8`, which adds the mapping $\mathtt{z} \to x - 8$ to the symbolic store.

Then, when execution reaches branch `z < 4`, we discover that under the current PC both branches are feasible, and we fork execution again, adding the constraint $x - 8 < 4$ on the `then` side, and the constraint $x - 8 \geq 4$ on the `else` side. The latter path terminates immediately by executing `return 0`, while the former executes `arr[z] = 'A'`. Prior to this array indexing instruction, symbolic execution inserts an implicit check asking if the array index is guaranteed to be in bounds. On this path, the PC is $x - 1 > 7 \wedge x - 8 < 4$, which can be used to establish that $\mathtt{z} \to x - 8$ cannot be out of bounds.

In shadow symbolic execution, our goal is to generate inputs that trigger the new behaviours introduced by a patch. While various definitions of *behaviour* are possible (especially if higher-level semantic information about the program is available), in this paper we use a generally-applicable definition of behaviour at the code-level: the behaviour of the program on a certain input is represented by the sequence of edges in the control-flow graph of the program traversed during execution. We say that two versions *diverge* on an input if their code-level behaviours are different for that input. Note that a code-level divergence may or may not result in an observable *output difference*.

To find inputs exposing different behaviour across versions, we start by executing both the old and the new version of the program on an input that exercises the patch, and gather constraints on the side, as in the dynamic symbolic execution variant called *concolic execution* [9, 27]. Until the patch is reached, assuming deterministic code, both the symbolic stores and the path conditions are identical for the two versions (by definition, since they have yet to execute a different instruction). However, once the patch is reached, the two versions might update their symbolic stores and path conditions differently. In our approach, we let each version update its symbolic store as required, sharing the two stores efficiently (see §3.2).

```
1  char arr[4];
2
3  int foo(int x) {
4    int y = change(x - 1, x + 1); // y=x-1 -> y=x+1
5    if (y > 7) {
6      int z = x - 8;
7      if (z < 4)
8        arr[z] = 'A';
9      return 0;
10   }
11   return 1;
12 }
```

Figure 3: A toy example showing a simple patch that modifies an `if` statement.

When a branch condition is reached, we evaluate it under the symbolic stores of each version, and we explore the entire branch cross product. Figure 2 illustrates the general case, where we reach a branch condition that evaluates to semantically-different expressions in the two versions—say, *old* in the old version, and *new* in the new version. Instead of forking execution into two paths (if possible) based on the execution of the new version—one adding the condition *new* and the other $\neg new$, we fork into up to four ways. On two of these cases the two versions behave identically (denoted by *same* in the figure): both versions take either the `then` ($new \wedge old$) or the `else` ($\neg new \wedge \neg old$) branch. On the other two, the executions of the two versions diverge (denoted by *diff* in the figure): either the new version takes the `then` branch and the old version the `else` branch ($new \wedge \neg old$), or the new version takes the `else` branch and the old version the `then` branch ($\neg new \wedge old$).

There are two scenarios of interest whenever the initial input reaches such a branch:

(1) ***Concrete executions diverge.*** That is, the input makes the two program versions follow different sides at this branch. This means that developers have already done a good job exploring at least part of the new behaviour introduced by the patch. However, this one input might not be sufficient to explore all the new behaviours—for example, the new version might go on and execute a lot of new code introduced by the patch. To better test the patch, at this point we enable a *bounded symbolic execution* run on the new version, i.e. we start symbolic execution in a breadth-first search mode, for a fixed time budget. This also lets us generate other divergent inputs exhibiting the same branching behaviour up to that point (but different afterwards).

(2) ***Concrete executions are identical, but divergences are possible.*** That is, the input makes the two programs take the same side of the branch, but at least one of the *diff* paths in Figure 2 is feasible. In this case, we also explore those paths. For each feasible *diff* path, we first generate an input that exercises the divergent behaviour, and then continue doing bounded symbolic execution in the new version in order to systematically and comprehensively explore additional divergent behaviours.

As long as the concrete executions do not diverge, we continue running both versions until the end of the program, exploring any additional possible divergences along the way.

Toy example. As an illustrative example, consider again the code in Figure 1, and assume that the developers have written a patch that changes `y = x - 1` to `y = x + 1`. We repeat for convenience the code in Figure 3, where the changed code is marked using the annotation `change()`. Furthermore, suppose that the developers have written three test cases to exercise the patch: `x = 0`, `x = 9` and `x = 15`. These tests achieve full branch coverage in both versions, but fail to exercise the new behaviour introduced by the patch and miss a buffer underflow bug introduced for `x = 7`.

Shadow symbolic execution provides a systematic way of testing the new behaviours introduced by a patch. Its effectiveness and performance depend on the starting input that touches the patch, but in our example, it can find the bug starting from any of the three inputs, with similar amount of effort. We illustrate how it works starting from input `x = 0`. When function `foo` is entered, both symbolic stores are $\{\mathtt{x} \rightarrow x\}$ and the PC is `true`. After the patched code on line 4 is executed, the symbolic stores become $\{\mathtt{x} \rightarrow x, \mathtt{y} \rightarrow x - 1\}$ in the old version and $\{\mathtt{x} \rightarrow x, \mathtt{y} \rightarrow x + 1\}$ in the new version. As a result, when line 5 is reached, the condition `y > 7` evaluates to $x - 1 > 7$ in the old version, and to $x + 1 > 7$ in the new version. At this four-way fork, our input `x = 0` follows one of the *same* cases illustrated in Figure 2. However, both *diff* cases are also feasible at this point, so shadow symbolic execution first generates an input that triggers the divergent behaviour in each case, and then starts from that point a bounded symbolic execution run on the new version.

One *diff* case, when at line 5 the old version takes the `else` side while the new version takes the `then` side, generates the condition $x - 1 \leq 7 \wedge x + 1 > 7$. At this point, the constraint solver may return $x = 7$, which exposes the buffer underflow bug, but it could also return $x = 8$, which does not. In both cases, we start bounded symbolic execution on the new version, which finds the bug, thanks to the implicit index-in-bounds check injected by the symbolic execution engine before each array access. Note that the bounded symbolic execution phase is started only on the divergent path (in our case when the new version takes the `then` side on line 5) and with the path condition that triggers the divergence (in this case $x - 1 \leq 7 \wedge x + 1 > 7$). This significantly constrains the search space, making symbolic execution explore *only paths that expose new behaviours introduced by the patch.*

While not relevant for our buffer underflow bug, note that the patch also introduces a divergence which causes the old version to take the `then` side and the new version the `else` side at line 5, resulting in a divergence condition $x - 1 > 7 \wedge x + 1 \leq 7$. This divergence is less obvious because it only occurs when there is an arithmetic underflow on line 4. For example, when x is -2147483648, y becomes 2147483647 in the old version,[1] and -2147483647 in the new version, causing the unexpected divergence. The subtle point is that $x - 1 > 7$ does not imply $x > 8$ for fixed-width machine arithmetic, which illustrates the difficulty of manually reasoning about the new behaviours introduced by software patches and the need for automatic techniques to help in the process.

3. SHADOW SYMBOLIC EXECUTION

Figure 4 presents an overview of the process of testing software patches with shadow symbolic execution. The inputs to our technique are: i) the old and the new version of the

[1] In `gcc 4.8.2`; signed overflow is undefined in C.

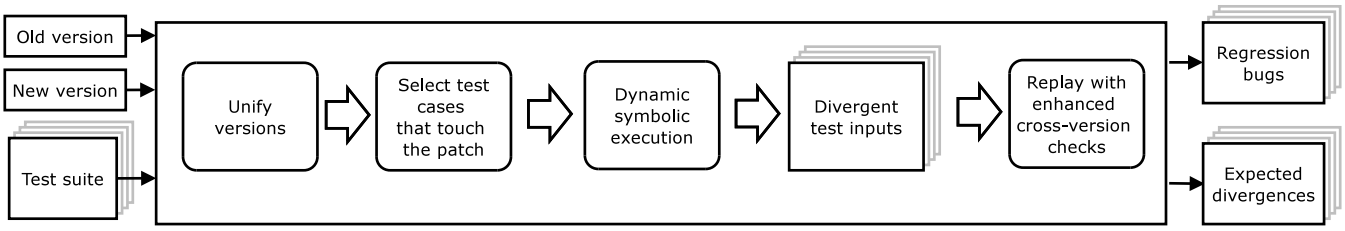

Figure 4: A high-level overview of shadow symbolic execution.

program under test (alternatively, the old version and the patch), and ii) the program's test suite. The output is a set of inputs that expose divergent behaviour between versions, triggering either regression bugs or expected divergences. We further divide these divergent behaviours into four subcategories. First, divergences that lead to generic errors (e.g. memory errors) only in the new version are clear regression bugs that should be fixed. Second, divergences that lead to generic errors only in the old version are expected divergences that witness the fix of that error. Third, divergences that propagate to the output are of interest to developers, because they can be used to quickly assess whether they are intended changes or regression errors. Finally, divergences that do not lead to any noticeable differences could still be of interest to developers, who could add the corresponding inputs to the application's test suite.

In the first step of our approach, we annotate the patches as illustrated in Figure 3, in order to unify the old and the new version into a single program that incorporates them both (§3.1). Next, we select from the test suite those test cases that touch the patch. We then perform shadow symbolic execution and generate inputs that expose divergent behaviour (§3.2). Finally, we run both versions natively on all divergent inputs using enhanced cross-version checks and identify those that trigger errors or output differences (§3.3).

3.1 Unifying versions via patch annotations

Our approach to executing both the old and the new version of the program in the same symbolic execution instance is to enforce them to proceed in lockstep until they diverge in control flow. This is done by creating a single *unified program* in which the two versions are merged via `change()` annotations, as we have shown on line 4 in Figure 3. Mapping program elements across versions [15] is a difficult task, as in the extreme, the two versions could be arbitrarily different programs. However, in practice the process can be made sufficiently precise and furthermore automated using various heuristics, as shown by recent work [16, 23].

We currently add these annotations manually, following the annotation patterns discussed below; however, we believe many patterns could be applied automatically, although we leave this for future work.

Our annotations use the macro `change()`, which resembles a function call with two arguments: the first argument represents the code expression from the old version and the second argument the corresponding expression from the new version. One key property is the ability to run the old version by replacing `change()` with its first argument, and the new version by replacing it with its second argument.

Writing these annotations was easier than we initially expected—we started by targeting very small patches (1-2 lines of code), but ended up annotating large patches of up to several hundred lines of code. Below, we discuss the main annotation patterns that we follow, in the order in which we typically apply them.

1. *Modifying an rvalue expression.* When an expression `E1` is changed to `E2`, the annotation is simply `change(E1, E2)`. As a general principle, we always push the `change()` annotations as deep inside the expression as possible. This strategy optimises the sharing between the symbolic stores of the two versions, and it also allows for various optimisations, such as constant folding, to be performed by the symbolic execution engine. Examples include:

   (a) *Changing the right-hand side of an assignment:*

   ```
   x = y + change(E1, E2);
   ```

   (b) *Changing an argument in a function call:*

   ```
   f(..., change(E1, E2) + len(s), ...);
   ```

   (c) *Changing a conditional expression:*

   ```
   if (change(E1, E2))
     ... code ...
   ```

   In the patches we examined, we observed that developers often change the control flow in the program by strengthening or weakening existing conditional expressions, i.e. by adding or removing boolean clauses. For instance:

   (d) *Weakening a condition from A to A || B*:

   ```
   if (A || change(false, B))
     ... code ...
   ```

   (e) *Strengthening a condition from A to A && B*:

   ```
   if (A && change(true, B))
     ... code ...
   ```

   We choose a different style of annotations for strengthening of a condition from A || B to B and for weakening a condition from A && B to B:

   (f) *Strengthening a condition from A || B to B*:

   ```
   if (change(A || B, B))
     ... code ...
   ```

   (g) *Weakening a condition from A && B to B*:

   ```
   if (change(A && B, B))
     ... code ...
   ```

   The reason for using this different style is to avoid the introduction of spurious divergences. For example, if we annotated a strengthening of a condition from A || B to B as `if (change(A, false)|| B)`, then if A is `true` and B is also `true`, a divergence would be reported, even though the two versions would take the same `then` side of the branch. While this annotation might be preferable when a stronger coverage criterion such as MC/DC [11] is desired, in our experiments we prioritise divergences that propagate to the output.

```
1 x = change(a, b);
2 y = x + 1;
3 z = y / 2;
4 ...
5 if (z) {
6   ... code ...
7 }
```

Figure 5: A change in an assignment propagating through the rest of the code.

2. *Adding/removing extra assignments or conditionals.* Essentially, we view all changes of this type as modifications of existing constructs by adding *dummy* statements at appropriate points in the program [26]. E.g.:

   (a) *Adding an extra assignment* `x = E`:

   ```
   x = change(x, E);
   ```

   (b) *Removing an assignment* `x = E`:

   ```
   x = change(E, x);
   ```

   (c) *Adding code conditional on an expression.* That is, if the code added in the new version has the form `if (C)... code ...` , the annotation is:

   ```
   if (change(false, C))
     ... code ...
   ```

   (d) *Removing code conditional on* C:

   ```
   if (change(C, false))
     ... code ...
   ```

3. *Adding/removing straightline code fragments.* In general, we first try to annotate any code modifications using rules 1 and 2. However, if the changed code has side effects (e.g. it writes to a file) or the previous rules are too difficult to apply, we use the following rules:

   (a) *Removing straightline code:*

   ```
   if (change(true, false))
     ... code ...
   ```

   (b) *Adding straightline code:*

   ```
   if (change(false, true))
     ... code ...
   ```

   We note that this is the most conservative way of annotating a change in our framework—the execution of a branch instruction conditional on a `change(true, false)` expression immediately triggers the generation of a divergent test input, terminates shadow execution and proceeds by running the new version only, losing the ability to use the old version as an oracle.

4. *Adding/removing variable declarations.* If a variable declaration is added or removed, we keep it in the merged program; no annotations are necessary. Uses of that variable are treated using rules 1 to 3 above.

5. *Modifying variable declarations.* When the type of a variable is changed to include more or fewer values, we keep the larger type. Due to arithmetic overflow issues, we reason manually whether this is safe to do; however, in our benchmarks type changes were a rare occurrence and quite straightforward: e.g. changing `char buf[5]` to `char buf[2]` or changing a `bool` to an `enum`.

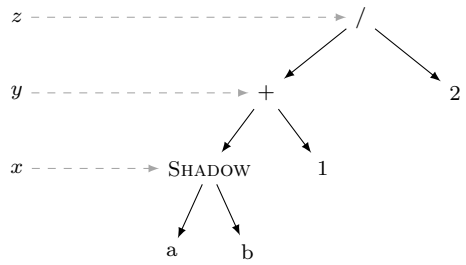

Figure 6: A shared expression tree for the expressions corresponding to the variables `x`, `y` and `z` in Figure 5. Expressions containing shadow subexpressions are kept in the symbolic store and lazily evaluated at symbolic branch points, e.g. in the `if` condition on line 5, in order to extract the new and old counterparts.

3.2 Symbolic execution phase

For each input that touches the patch, shadow symbolic execution operates in two phases:

1. **Concolic phase.** We start by executing the unified program on that input, and gather constraints on the side, as in concolic execution [9, 27]. As the program executes:

   (a) If at a branch point the input exposes a divergence, we stop shadow execution and add this divergence point to a queue to be processed in phase 2.

   (b) If at a branch point the input follows the same path in both versions, but divergences are also possible, we generate a test case exposing each possible divergence and then add these divergence points to the queue to be processed in phase 2. We then continue the concolic execution of the unified program.

2. **Bounded symbolic execution (BSE) phase.** For each divergence point placed in the queue, we initiate a BSE run in the new version starting from that divergence point, to search for additional divergent behaviours.

The concolic phase is computationally cheaper; nevertheless, the BSE phase is essential as it is able to propagate the divergent behaviour down the execution tree and explore systematically the impact of the divergence.

Efficiently sharing state using shadow expressions. As in other instances when different software variants or versions are run together [8, 12–14, 22, 30], shadow symbolic execution can substantially increase memory consumption. As a result, it is important to maximise sharing between the symbolic state of the two versions. Since the patch typically affects a relatively small number of symbolic expressions, everything else can be shared. Furthermore, it is possible to share those parts of symbolic expressions that are identical between versions.

To enable sharing, whenever we encounter a `change()` annotation, instead of constructing and maintaining separate symbolic expressions for the old and the new version, we create a *shadow expression*. A shadow expression contains two subexpressions, one corresponding to the old version, and one to the new. Shadow expressions can be used as any other expressions, without the need to duplicate for each version entire expression trees that contain modified subexpression

nodes. To illustrate, consider the example in Figure 5, in which the code is changed to assign into x value b instead of a. Furthermore, assume that after this change x is used multiple times in the program, directly or indirectly, e.g. to derive variables y and z on lines 2 and 3. Without sharing, both y and z would have to point to different symbolic expressions in the two versions. However, the use of shadow expressions unifies the expressions for the two versions and maximises sharing. In our example—as illustrated in Figure 6—x will point to a shadow expression with children a and b. Then, when y is created, its left child is assigned to this shadow expression, but node y itself remains the same in both the old and the new versions. Similarly, when z is created, its children become simply y and 2. This scheme has the advantage that sharing is maximised, propagation of changes is implicit, and the creation of expressions can still be performed in constant time.

In addition, the dynamic nature of symbolic execution provides opportunities for identifying refactorings on a per-path basis at run time, which allows for further optimisations. In particular, if the two candidate children e_{old} and e_{new} of a shadow expression are equivalent under the current PC, then the syntactic changes do not introduce semantic differences and we skip the creation of a shadow expression.

3.3 Enhanced cross-version checks

In order to determine whether an input that exposes a code-level divergence results in an externally-observable regression bug or expected behavioural change, we use a series of enhanced cross-version checks. These checks run the two versions natively on each input which exposes a divergence and compare their outputs, including exit codes. They also check for generic errors, in particular crashes and memory errors that do not trigger a crash, the latter detectable by compiling the code with *address sanitization* [28].

If the outputs of the two versions differ, it is up to developers to decide whether the difference is expected or a regression bug. Even though in our evaluation we determined this automatically (because we also knew the patches that fixed the introduced bugs), we validated the classification manually and often found making this judgement easy to do, by reading the commit message describing the intention of each patch. We also had cases in which it was not immediately obvious whether the change in behaviour was expected—these are exactly the kind of inputs that developers should pay attention to, as they could point to bugs or lack of proper documentation.

We apply these checks both on the inputs in the regression test suite and on those generated by our technique.

4. IMPLEMENTATION

We implemented our approach in a tool called SHADOW, which is built on top of the KLEE symbolic execution engine [5] and uses the concolic execution functionality from the ZESTI extension [19]. Our code is based on KLEE revision 02fa9e4d, LLVM 2.9 and STP revision 1668.

To select the test cases in the regression suite that touch the patch, we run the regression suite on the new version of the program compiled with coverage instrumentation (gcov[2]).

To run the concolic phase, we replace the program under test with a wrapper script that passes the original invocation parameters to SHADOW. Note that a test case may invoke a given program multiple times. To test the n^{th} invocation, the script runs the first $n-1$ invocations natively, and forwards the n^{th} to SHADOW.

The concolic phase runs each test case touching the patch, and generates a test input every time it finds a divergence. The BSE phase repeats the concolic phase (for ease of implementation) and stores all divergence points in a queue. Then, SHADOW performs bounded symbolic execution starting at each divergence point in this queue, in a breadth-first search manner. We generate an input for each path explored during BSE. As these paths originate from divergent points, by definition all generated inputs expose divergences.

In the concolic phase, a single invocation is allowed to run for a maximum of 600s. The same budget is given for the BSE phase, which as discussed above, also repeats the concolic phase for ease of implementation. The actual symbolic exploration phase is given 570s, divided equally among all the divergence points placed in the queue.

For each phase, we set a global timeout of 3600s for running an entire test case (potentially consisting of multiple invocations). Also for each phase we set a global timeout of 7200s for running all the test cases that touch the patch.

In order to rerun the generated inputs natively, SHADOW provides a *replay* functionality that implements the enhanced checks described in §3.3. This functionality also uses a wrapper script that calls the native versions of the application and substitutes the original parameters with the ones synthesised for the generated test cases. SHADOW runs each test input twice, once with the old version and once with the new one. The outputs and exit codes are then compared in order to discover divergences that propagate to the program output. For each phase, the replay is bounded by a per-invocation timeout of 5s, a per-test-case timeout of 60s and a global timeout of 7200s. Due to some non-determinism in our replay infrastructure we retry each replay experiment once if the first attempt is unsuccessful.

5. EVALUATION

We evaluate SHADOW on the software patches from the GNU Coreutils application suite[3] included in the CoREBench suite of regression bugs.[4] Coreutils is a collection of utility programs for file, text and shell manipulation. It is a mature, well-maintained and widely-used project included in virtually all Linux distributions. Together, the programs form a code base of over 60 KLOC.[5]

The CoREBench patches represent a tough challenge for test input generation: as the CoREBench authors discuss [4], the complexity of these regression errors and associated fixes is significantly higher than those in popular evaluation suites such as the SIR and Siemens benchmarks.[6]

CoREBench provides 22 pairs of {*bug-introducing, bug-fixing*} patches for Coreutils. However, some patches introduce multiple bugs, so we are left with 18 unique bug-introducing patches, which are shown in Table 1. The first column of this table shows the CoREBench ID for the patch. If the patch is responsible for multiple bugs, we show all relevant IDs: e.g., "5 = 16" means that the bug-introducing patch with ID 5 is

[2]https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[3]http://www.gnu.org/software/coreutils/

[4]http://www.comp.nus.edu.sg/~release/corebench/

[5]Measured with the cloc tool, http://cloc.sourceforge.net/

[6]http://sir.unl.edu/portal/index.php

Table 1: `Coreutils` patches from `CoREBench`. We report the patch size (which takes into account source code only), the number of test files that touch the patch, and the number of `change()` annotations that we used for each patch.

ID	Tool	Patch Size			Test Files Touching	Annotations
		LOC	Hunks	Files		
1	mv, rm	45	17	4	243	12
2	od	141	46	1	–	32
3	cut	294	35	1	17	14
4	tail	21	4	1	4	4
5=16	tail	275	13	1	2	1
6	cut	8	3	1	15	3
7	seq	148	5	1	29	5
8	seq	37	4	1	29	12
9=18=20	seq	324	52	1	–	11
10	cp	16	8	5	42	2
11	cut	2	1	1	14	1
12=17	cut	110	17	1	1	4
13	ls	13	2	1	8	2
14	ls	15	5	1	7	4
15	du	3	1	1	26	1
19	seq	40	9	1	11	6
21	cut	31	10	1	11	6
22	expr	54	6	1	2	4

Table 2: Distribution of annotation patterns across the `CoREBench` patches. The last two columns refer to changes that require no explicit annotations despite taking some effort to reason about. Variables that are added or removed in a scope accessible only to a single version (e.g., in a newly-added function), do not contribute toward column 5.

ID	Modified rvalues	Extra assign/cond	Straightline code	Added/removed variables	Modified types
1	11	1	-	-	✓
2	24	-	8	✓	✓
3	4	6	4	✓	-
4	3	1	-	-	-
5=16	-	1	-	✓	-
6	1	2	-	-	-
7	2	3	-	✓	-
8	11	1	-	-	-
9=18=20	9	2	-	✓	✓
10	-	2	-	✓	-
11	-	1	-	-	-
12=17	2	-	2	✓	✓
13	1	-	1	-	-
14	3	-	1	✓	-
15	-	1	-	-	-
19	2	3	1	-	-
21	4	2	-	✓	-
22	3	1	-	✓	-
Total	80	27	17	✓	✓

the same as the one with ID 16. The *LOC*, *Hunks* and *Files* columns provide information about the size of each patch, in terms of added/modified lines of code (LOC), hunks and files. The number of LOC is measured by the `diff` tool. The *hunks* forming a patch are essentially the different areas of code affected by a patch. More formally, a hunk groups together all the lines added or modified in a patch which are at a distance smaller than the context size. We used the unified diff format with a context size of zero when computing the hunks. As can be seen, the size of a patch varies between only 2 LOC and a single hunk and up to 324 LOC and 52 hunks. Most patches change a single file, with two exceptions, where 4 and respectively 5 different files are affected.

Finally, the column *Test Files Touching* gives the number of test files in the regression test suite that touch the patch, which are used as starting points by SHADOW. Due to technical problems related to running old revisions, we could not run the test suites coming with patches #2 and #9. Therefore, we exclude those two patches from our evaluation, although we do report the annotation effort involved for these patches too.

5.1 Annotations

Column *Annotations* in Table 1 shows the number of `change()` annotations that we added for each patch. In general, the number of annotations does not depend on the number of LOC in the patch. For example, patch #5 adds a call to a new function consisting of over 200 LOC, which in turn calls other new code. However, while a lot of code has been added, we need a single `change()` annotation to enable it, as discussed in §3.1.

Instead, the number of hunks can give a rough estimate of the number of required annotations. Nonetheless, there are exceptions—for example, many hunks do not require any annotations. E.g., patch #5 discussed earlier includes a variable renaming from `nfiles` to `n_files` which results in many hunks that do not require any annotations. Hunks that only change comments are another example.

Table 2 provides a rough overview of the distribution of annotation patterns as classified in §3.1. The classification is approximate—for example, a transformation of a variable `bool neg` into `int sign` can be interpreted as both a change of type and an addition and a removal of a variable.

In general, there is often more than one way to annotate a patch. Furthermore, our manual effort is error-prone, although we are confident that the annotations are correct. We make our annotations publicly available,[7] hoping they will prove valuable in other differential testing projects too.

5.2 Experimental details

Environment. We conducted our experiments on a server running Ubuntu 14.04, equipped with two Intel(R) Xeon(R) E5-2450 v2 at 2.5 GHz CPUs (32 cores) and 192GiB of RAM. The tests were usually run in parallel for all the tested revisions.

Memory limit. We use KLEE's default memory limit of 2000 MiB per invocation, which was never exceeded.

Changes to code and test suites. Since some of the tested `Coreutils` revisions are several years old, they do not compile out of the box, and we had to apply several patches provided by the `CoREBench` authors. Furthermore, we had to make other minor modifications for compatibility with KLEE and our infrastructure.

To consistently compare the program outputs across versions, we also applied a series of changes to the `Coreutils` test suite related to making tests run (more) deterministically. One example is the creation of temporary files, which by default have different names across runs.

5.3 Overall results

We conduct three sets of experiments, corresponding to running (1) the regression test suite, (2) the concolic phase of SHADOW, and (3) the BSE phase of SHADOW. We run all three sets of experiments with our enhanced cross-version checks of §3.3. Note that for running the regression test suite with the enhanced checks, we use the same unified programs employed by the concolic and BSE phases of SHADOW. We take a conservative approach and assume that all invocations

[7] http://srg.doc.ic.ac.uk/projects/shadow

Table 3: Experimental results for the `Coreutils` patches in `CoREBench`, showing the number of code-level divergences detected, the percentage of these replayed, and the number of observed output differences, divided into expected changes and regression bugs. The inputs created by SHADOW for patch #21, marked with *, also expose an additional bug (an abort) which is different from the one detected by the regression suite (wrong exit code).

ID	Regression Suite				Concolic Phase				BSE Phase				Total		
	Divergences		Differences		Divergences		Differences		Divergences		Differences		Divergences	Differences	
	Total	Replayed	Expected	Bug	Total	Replayed	Expected	Bug	Total	Replayed	Expected	Bug	Total	Expected	Bug
1	2,434	29%	3	-	-	-	-	-	37,533	27%	-	-	39,967	3	-
3	524	51%	-	-	-	-	-	-	15,110	7%	-	-	15,634	-	-
4	6	100%	6	-	-	-	-	-	33	100%	30	-	39	36	-
5=16	7	100%	-	2	-	-	-	-	7	100%	-	-	14	-	2
6	232	100%	-	-	15	100%	-	-	1,205	59%	-	86	1,452	-	86
7	62	100%	5	-	-	-	-	-	62	100%	-	-	124	5	-
8	24	100%	-	-	18	100%	-	-	54,800	6%	-	-	54,842	-	-
10	3	100%	-	2	-	-	-	-	3	100%	-	-	6	-	2
11	51	100%	9	-	6	100%	-	-	865	100%	-	-	874	9	-
12=17	163	100%	-	-	-	-	-	-	4,069	45%	-	78	4,232	-	78
13	7	100%	1	1	-	-	-	-	4	100%	-	-	11	1	1
14	2	100%	-	-	-	-	-	-	-	-	-	-	2	-	-
15	1	100%	1	-	-	-	-	-	-	-	-	-	1	1	-
19	66	100%	7	-	236	100%	-	-	33,015	27%	-	-	33,317	7	-
21	115	100%	12	4	348	100%	136	122*	20,745	5 %	3	558*	21,208	151	684
22	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

in the regression suite which execute the `change()` annotations with arguments of different value are divergent. Note that this is an over-approximation as the two versions might still have the same branching behaviour.

Regarding run times for the different phases of SHADOW, we observed median values of 1,020s for running the regression suite; 1,962s for running the concolic phase; and 7,213s for running the BSE phase, all including our enhanced checks. Note that these values are only meant to give a rough estimate of the time needed by SHADOW—they are influenced by the different numbers of test cases that touch the patch, and also by the load on our machine, which we have not tried to control.

Table 3 gives an overview of our experimental results. For each phase, we provide the number of test inputs exposing code-level divergences (*Divergences–Total*) and the percentage of those inputs that we manage to replay in the allotted time frame (*Divergences–Replayed*). Due to a small degree of non-determinism in our replay infrastructure, a few of these divergences might be duplicates. The figures under *Total–Divergences–Total* are an over-approximation, as they reflect our conservative approach of assuming that all invocations in the regression suite that touch `change()` annotations with different value arguments are divergent.

Table 3 also presents how many of the divergent inputs that we replayed led to output differences. These differences are further classified into *expected* differences and *bugs*. There are two types of bugs: generic bugs such as memory errors, and semantic bugs that lead to incorrect results. For generic bugs, if the old version does not trigger the error and the new one does, then we report the input as exposing a regression error. If it is the other way around, we report it as exposing an expected difference.

As discussed in §3.3, by reading the commit message associated with the patch, we can often reason manually whether an input that leads to different outputs across versions exposes a regression bug or an intended change in behaviour. We expect this would be even easier for the authors of those patches. However, in our evaluation we make use of the fact that the `CoREBench` regression suite provides the revisions fixing the introduced bugs. More precisely, we run the input on two additional versions: the version just before the fix and the version in which the fix was applied. If these two versions behave the same on this input or the fixed version behaves the same as the new version, we classify the change in behaviour as expected. Otherwise, we classify it as a regression bug. For patches introducing multiple bugs, we run each fix in turn. This approach is automatic, but is not guaranteed to correctly classify changes in behaviour due to: (1) non-determinism and (2) because the patch fixing the bug may introduce other changes too. Hence we also perform a brief manual sanity check of the automatic classification.

5.4 Successful examples

Table 4 gives several examples of actual inputs generated by SHADOW. For each input, we show side-by-side the behaviours of the old and the new version. For instance, the first example shows an expected difference in `tail`, while the second example shows a regression bug that triggers a buffer overflow in the new version of `cut`.

We discuss in more detail two patches in which we managed to find the introduced regression bug and/or the intended change in behaviour.

`CoREBench` patch #6. This is a patch in `cut`, a tool whose purpose is to delete portions of text (ranges of bytes, characters or fields) from each line of a file. To do so, the user can specify both *closed ranges*, e.g. `3-5`, meaning all the bytes (or characters or fields) from the third to the fifth byte from the beginning of the line, as well as *open ranges*, e.g. `-5` and `9-`, to refer to all the bytes (or characters or fields) from the beginning of the line up to the fifth byte, and from the ninth byte until the end of the line, respectively. The aim of the patch is to prevent unnecessary memory allocation when only open ranges are specified.

The annotated patch is presented in Figure 7. We added three annotations, one when an `if` statement is removed (line 1), one when an `if` statement is added (line 4), and one when an extra conjunct is added to the condition of an `if` statement (line 10).

Table 4: Sample inputs generated by SHADOW exposing regression bugs and expected differences.

ID	Generated Input	Behaviour		Classification
		Old	New	
4	`tail --retry ///s\x01\x00g\x00`	tail: warning: --retry is useful mainly when following by name...	tail: warning: --retry ignored; --retry is useful only when following...	Expected
6	`cut -c1-3,8- --output-d=: ⟨file⟩` *file contains "abcdefg"*	abc	abc + *buffer overflow*	Bug
17	`cut -c1-7,8- --output-d=: ⟨file⟩` *file contains "abcdefg"*	abcdefg	abcdefg + *buffer overflow*	Bug
21	`cut -b0-2,2- --output-d=: ⟨file⟩` *file contains "abc"*	abc	*signal abort*	Bug
21	`cut -s -d: -f0- ⟨file⟩` *file contains ":::\n:1"*	:::\n:1	\n\n	Expected
21	`cut -d: -f1,0- ⟨file⟩` *file contains "a:b:c"*	a:b:c	a	Expected

```
1  if (change(max_range_endpoint < eol_range_start,
       false))
2    max_range_endpoint = eol_range_start;
3  ...
4  if (change(true, max_range_endpoint))
5    printable_field = xzalloc (max_range_endpoint /
         CHAR_BIT + 1);
6  ...
7  if (output_delimiter_specified
8      && !complement
9      && eol_range_start
10     && change(true, max_range_endpoint)
11     && !is_printable_field (eol_range_start))
12   mark_range_start (eol_range_start);
```

Figure 7: CoREBench bug #6.

Prior to the patch, memory was allocated unconditionally (line 5), but the patch strengthened the condition guarding the allocation based on the value of `max_range_endpoint`, which represents the maximum end value of an index in a closed range, and is 0 when the user specifies only open ranges. The patch introduces a buffer overflow on line 11, when both closed and open ranges are specified and the value of `max_range_endpoint` is greater than 0 but smaller than the minimum start value of an index in an open range (`eol_range_start`). In such cases, `max_range_endpoint` in the new version is not set to the value of `eol_range_start` (line 2) and on line 5 the `printable_field` array is allocated to size `max_range_endpoint + 1`. Finally, on line 11, function `is_printable_field()` accesses the `printable_field` array at index `eol_range_start` which results in an index out-of-bounds error in the new version.

Table 4 shows a test input generated by SHADOW that exposes this bug. The input was found in the BSE phase.

CoREBench patch #21. This patch intends to make `cut` emit an error message when invalid ranges such as `2-0` are specified.

Our annotated patch is given in Figure 8. We added six annotations: three of them when an rvalue expression is changed (lines 3, 4 and 15), one which adds a new `if` statement (line 9), one which removes an `if` statement (line 17), and one which modifies an `if` statement (line 12).

The test cases in the regression suite already detect 12 expected output differences exposing the same behaviour in which the new version prints out one of the two error messages on lines 10 and 15. However, SHADOW generated further inputs for which the output differences do not involve error messages. The last two rows of Table 4 show two such inputs. Using the bug-fixing revision, we classified these changes as expected, and we think the generated inputs are good candidates for being added to the regression suite.

```
1  @@ set_fields (const char *fieldstr)
2    ...
3    initial = change(value, lhs_specified) ? value : 1;
4    value = change(value, true) ? 0 : value;
5  }
6  else if (*fieldstr == ',' || isblank (*fieldstr) ||
       *fieldstr == '\0') {
7    ...
8    dash_found = false;
9    if (change(false,!lhs_specified && !rhs_specified))
10     FATAL_ERROR (_("invalid range with no endpoint: -
           "));
11   ...
12   if (change(value == 0, !rhs_specified)) {
13     ...
14     if (value < initial)
15       FATAL_ERROR (_(change("invalid byte or field
             list", "invalid decreasing range")));
16   ...
17   else if (change(value != 0, true)) {
18     ...
```

Figure 8: CoREBench bug #21.

SHADOW also found unexplored divergences just off the paths executed by the test suite, which revealed an abort failure. A sample such input generated by SHADOW during the concolic phase is `-b0-2,2- --output-d=: file`. In the BSE phase, SHADOW detected a buffer overflow bug similar to the one discussed in patch #6. Note that these are separate bugs from the one recorded in CoREBench.

5.5 Unsuccessful executions

For several patches SHADOW failed to synthesise inputs that trigger either expected divergences or bugs. Regarding expected divergences, we note that several patches seem to be refactorings, so it would be impossible to trigger an expected output difference (any difference would be a bug).

In terms of the missed regression bugs, as mentioned before, the CoREBench patches are very challenging, and significantly more complex than those typically considered by prior research studies—see the CoREBench paper for details [4]. To get a feel for the challenges involved in analysing these patches, consider the following bugs missed by SHADOW: finding

bug #1 requires reasoning about file access rights, bug #8 requires floating point support, bug #14 requires support for symbolic directories, and the bug report for #19 is not reproducible on our recent distribution of Linux. Finally, our relatively short timeout values may have prevented us from successfully detecting some of the bugs and expected divergences; we chose these values to keep the turnaround time for running all experiments within a nightly run.

More generally, some of these patches require a precise environmental model (KLEE's model is incomplete, e.g., lacks the ability to handle symbolic directories), and at least one requires support for symbolic floating-point values (which KLEE does not provide). We also depend on the quality of the inputs in the test suites from which we start exploration.

Our mechanism for detecting changes is also limited, focusing solely on output differences. However, some patches change non-functional properties such as improving memory consumption in #6 or performance in #7.

Finally, note that with one exception, we always have inputs that expose divergences at the code level, which could prove useful to developers to reason about their patches. However, in many cases the number of divergent inputs is simply too large, and in future work we plan to investigate clustering and ranking techniques to help developers sift through these divergences.

5.6 Reflections on regression testing process

Our experience with the `CoREBench` patches revealed several insights into the regression testing process. First, we believe that cross-version checks could be easily incorporated into existing regression test suites. We envision a process in which developers would examine divergent inputs and confirm whether the change in behaviour is expected or not. Such a lightweight process would have detected some of the complex regression bugs in `CoREBench`. Second, generating inputs that trigger externally-visible differences is valuable both for the possibility of finding regression bugs, as well as for documentation—regarding the latter, we found that such inputs are often the best "explanation" of the patch.

6. RELATED WORK

We introduced the high-level idea behind shadow symbolic execution in a short idea paper [6], but without any implementation or evaluation.

Recent years have seen a lot of work on automatic techniques for testing software patches, with many of these techniques based on symbolic execution [1,3,17,18,20,25,29,31]. However, most research efforts have looked at the problem of generating test inputs that cover a patch. By contrast, input generation targeting behavioural changes introduced by a patch has received much less attention.

Differential symbolic execution [24] is a general framework that can reason about program differences, but its reliance on summaries raises significant scalability issues.

Directed incremental symbolic execution [25] combines symbolic execution with 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.

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.

The techniques discussed above were evaluated on patches significantly less complex than the `Coreutils` patches we considered. However, our technique is not fully automatic; while most of the annotations that we added could be automated, manual assistance might still be needed. Nevertheless, research on automating this step is promising [16, 23]; furthermore, note that even an imprecise automatic annotation system might be enough to help our technique generate inputs exposing behavioural changes.

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

Running the two program versions in the same symbolic execution instance is similar in spirit to running multiple versions in parallel, which has been employed in several other contexts, including online validation [22, 30], model checking [8], product line testing [14], and software updating [12].

Research on test suite augmentation requirements has used the differences between two program versions to derive requirements that test suites have to meet in order to ensure proper patch testing [10, 26]; our analysis could potentially provide further information to guide these techniques.

7. CONCLUSION

In this paper we have presented shadow symbolic execution, a novel technique for generating inputs that trigger the new behaviours introduced by software patches. The key idea behind shadow symbolic execution is to run both versions in the same symbolic execution instance, and systematically test any encountered code-level divergences. The technique unifies the two program versions via `change` annotations, maximises sharing between the symbolic stores of the two versions, and focuses exactly on those paths that trigger divergences. We implemented this technique in a tool called Shadow, which we used to generate inputs exposing several bugs and intended changes in complex `Coreutils` patches. We make our experimental data available via the project webpage at http://srg.doc.ic.ac.uk/projects/shadow.

8. ACKNOWLEDGEMENTS

We would like to thank Paul Marinescu and Daniel Liew for their assistance during this project, and the anonymous reviewers, Alastair Donaldson, Daniel Liew and Andrea Mattavelli, for their feedback on the paper. This research is generously supported by EPSRC through an Early-Career Fellowship and grant EP/J00636X/1 and by Microsoft Research through a PhD scholarship.

9. REFERENCES

[1] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *Proc. of the International Symposium*

on Software Testing and Analysis (ISSTA'11), July 2011.

[2] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Partition-based regression verification. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)*, May 2013.

[3] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Regression tests to expose change interaction errors. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, Aug. 2013.

[4] M. Böhme and A. Roychoudhury. Corebench: Studying complexity of regression errors". In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'14)*, July 2014.

[5] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Dec. 2008.

[6] C. Cadar and H. Palikareva. Shadow symbolic execution for better testing of evolving software. In *Proc. of the 36th International Conference on Software Engineering, New Ideas and Emerging Results (ICSE NIER'14)*, May 2014.

[7] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the Association for Computing Machinery (CACM)*, 56(2):82–90, 2013.

[8] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'07)*, July 2007.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*, June 2005.

[10] R. Gupta, M. J. Harrold, and M. L. Soffa. Program slicing-based regression testing techniques. *Software Testing Verification and Reliability (STVR)*, 6:83–112, 1996.

[11] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. A practical tutorial on modified condition/decision coverage. Technical Report NASA/TM-2001-210876, NASA, 2001.

[12] P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)*, May 2013.

[13] P. Hosek and C. Cadar. Varan the Unbelievable: An efficient N-version execution framework. In *Proc. of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, Mar. 2015.

[14] C. H. P. Kim, S. Khurshid, and D. Batory. Shared execution for efficiently testing product lines. In *Proc. of the 23rd International Symposium on Software Reliability Engineering (ISSRE'12)*, Nov. 2012.

[15] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proc. of the 2006 International Workshop on Mining Software Repositories (MSR'06)*, May 2006.

[16] W. Le and S. D. Pattison. Patch verification via multiversion interprocedural control flow graphs. In *Proc. of the 36th International Conference on Software Engineering (ICSE'14)*, May 2014.

[17] K.-K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks. Directed symbolic execution. In *Proc. of the 18th International Static Analysis Symposium (SAS'11)*, Sept. 2011.

[18] P. D. Marinescu and C. Cadar. High-coverage symbolic patch testing. In *Proc. of the 19th International SPIN Workshop on Model Checking of Software (SPIN'12)*, July 2012.

[19] P. D. Marinescu and C. Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)*, June 2012.

[20] P. D. Marinescu and C. Cadar. KATCH: High-coverage testing of software patches. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, Aug. 2013.

[21] P. D. Marinescu, P. Hosek, and C. Cadar. Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'14)*, July 2014.

[22] M. Maurer and D. Brumley. TACHYON: Tandem execution for efficient live patch testing. In *Proc. of the 21st USENIX Security Symposium (USENIX Security'12)*, Aug. 2012.

[23] N. Partush and E. Yahav. Abstract semantic differencing via speculative correlation. In *Proc. of the 29th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'14)*, Oct. 2014.

[24] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proc. of the ACM Symposium on the Foundations of Software Engineering (FSE'08)*, Nov. 2008.

[25] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'11)*, June 2011.

[26] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proc. of the 23rd IEEE International Conference on Automated Software Engineering (ASE'08)*, Sept. 2008.

[27] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, Sept. 2005.

[28] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, June 2012.

[29] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. eXpress: guided path exploration for efficient

regression test generation. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'11)*, July 2011.

[30] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *Proc. of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, Mar. 2009.

[31] Z. Xu and G. Rothermel. Directed test suite augmentation. In *Proc. of the 16th Asia-Pacific Software Engineering Conference (ASPEC'09)*, Dec. 2009.