Shadow Symbolic Execution for Testing Software Patches

TOMASZ KUCHTA, HRISTINA PALIKAREVA, and CRISTIAN CADAR,
Imperial College London

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 article, 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 input 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**; **Software evolution**; **Maintaining software**;

Additional Key Words and Phrases: Symbolic patch testing, regression bugs, cross-version checks

ACM Reference format:
Tomasz Kuchta, Hristina Palikareva, and Cristian Cadar. 2018. Shadow Symbolic Execution for Testing Software Patches. *ACM Trans. Softw. Eng. Methodol.* 27, 3, Article 10 (September 2018), 32 pages.
https://doi.org/10.1145/3208952

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, add new functionality, or refactor the code. 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 of the software.

Ideally, software changes, typically introduced by separate commits to the source-code repository and referred to as *patches*, should be comprehensively tested. At the very minimum, each line of code affected by the patch, i.e., appearing in the source-code diff between two consecutive commits, should be covered by at least one test case. While this level of testing is still far from

This research was generously supported by EPSRC through an Early-Career Fellowship and Grant No. EP/J00636X/1 and by Microsoft Research through a PhD scholarship.
Authors' addresses: T. Kuchta, H. Palikareva, and C. Cadar, Department of Computing, Imperial College London, Huxley Building, 180 Queen's Gate, London SW7 2AZ, United Kingdom; emails: {tomek.kuchta.mail, inanaiain}@gmail.com, c.cadar@imperial.ac.uk.

ACM 1049-331X/2018/09-ART10 $15.00
https://doi.org/10.1145/3208952

being achieved in practice [28], automatic techniques for enabling high-coverage patch testing are becoming more and more successful [1, 24, 27, 40, 43]. Many of these techniques are based on dynamic symbolic execution [8], 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 behaves identically for these inputs before and after the patch is applied. Careful analysis reveals 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 article, 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 the 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 work are:

(1) Shadow symbolic execution, a technique for generating inputs that trigger the new behaviours introduced by a patch. The technique effectively reduces the program search space by pruning a large number of irrelevant execution paths.
(2) A mechanism for unifying two program versions and representing 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.

A shorter version of this article was presented in [31], and an idea paper on this topic was published in [7]. In addition to providing more details and examples, this article presents an extended empirical evaluation that includes an investigation of how the time budget dedicated to the different stages of the technique impacts its overall effectiveness (Section 5.6), as well as a direct comparison with a vanilla symbolic execution approach (Section 5.7). It also includes a careful analysis of the limitations of the technique (Section 6.3) and a discussion of its applications to other programming languages (Section 6.4) and software engineering tasks (Section 6.5).

The rest of the article is organised as follows. We introduce shadow symbolic execution in Section 2 and then present it in detail in Section 3. We then give a brief overview of our prototype tool Shadow in Section 4 and describe our experience applying it to test a suite of complex patches in Section 5. We reflect on various aspects of shadow symbolic execution in Section 6, give an overview of related work in Section 7, and conclude in Section 8.

```
 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 }
```

Fig. 1. A toy example illustrating symbolic execution.

2 OVERVIEW

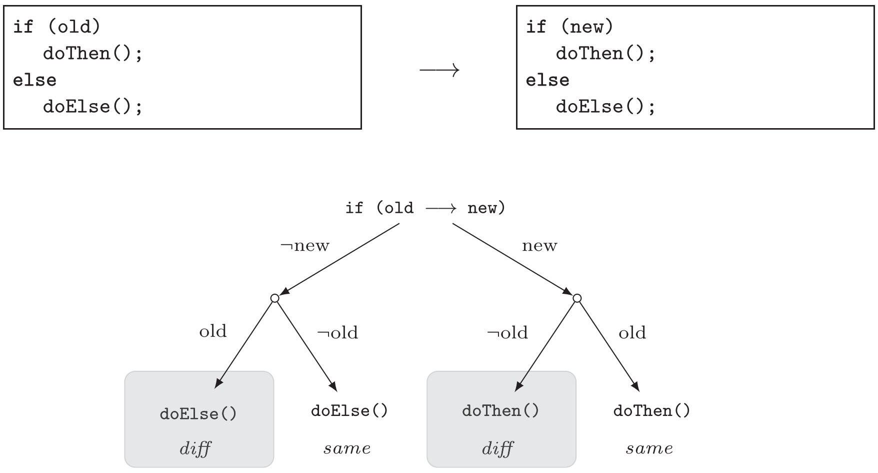
Shadow symbolic execution is meant to augment an existing test suite of a program. In our approach, we assume that we already have a test input that *touches* the patch, i.e., executes at least one patch statement—if such an input does not exist in the program's test suite, it could be generated using previous techniques such as KATCH [27], which uses targeted symbolic execution to explore paths that reach a patch.

Given such an input, our technique is designed to automatically generate new inputs that exercise the *new* behaviours introduced 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 [8], 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 and 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 the function foo on the symbolic input x. When symbolic execution starts, the symbolic store is $\{\texttt{x} \rightarrow x\}$, meaning that variable x maps to symbolic input x, and the PC is true. After line 4 is executed, the symbolic store becomes $\{\texttt{x} \rightarrow x,\ \texttt{y} \rightarrow x - 1\}$. When execution reaches branch y > 7, we establish by employing a constraint solver—typically a Satisfiability Modulo Theories (SMT) solver [11]—that under the current PC both 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, we add 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 $\texttt{z} \rightarrow x - 8$ 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 $x - 1 > 7 \wedge x - 8 < 4$, and the constraint solver can be used to establish that $\texttt{z} \rightarrow x - 8$ cannot be out of bounds.



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

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

Fig. 2. Four-way forking in shadow symbolic execution as a mechanism for capturing divergent executions (the ones shaded in grey).

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 article, 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 we gather constraints on the side, as in the dynamic symbolic execution variant called *concolic execution* [12, 38]. 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 Section 3.2).

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 when 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.[1] 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 ($\neg new \wedge old$).

[1]The expressions *old* and *new* are semantically different at a program location along a program path if they are not equivalent under the current path condition.

```
 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 }
```

Fig. 3. The toy example in Figure 1 with a simple patch that modifies the `if` statement on line 4.

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 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 $\{\texttt{x} \rightarrow x\}$ and the PC is `true`. After the patched code on line 4 is executed, the symbolic stores become $\{\texttt{x} \rightarrow x, \texttt{y} \rightarrow x - 1\}$ in the old version and $\{\texttt{x} \rightarrow x, \texttt{y} \rightarrow x + 1\}$ 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

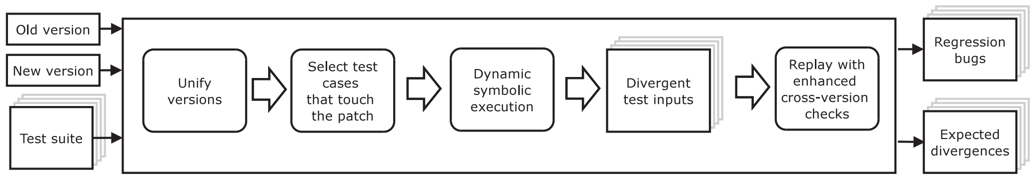

Old version
New version
Test suite
Unify versions
Select test cases that touch the patch
Dynamic symbolic execution
Divergent test inputs
Replay with enhanced cross-version checks
Regression bugs
Expected divergences

Fig. 4. A high-level overview of shadow symbolic execution.

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 divergence 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 a 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 that 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[2] 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 source code of the old and the new version of the 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 or assertion violations) 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 reflect 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.

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

In the first step of our approach, we annotate the patches as illustrated in Figure 3 to unify the old and the new version into a single program that incorporates them both (Section 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 (Section 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 (Section 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 [20] is a difficult task, as in the extreme, the two versions could be arbitrarily different programs. However, in practice the process can be streamlined using several annotation patterns, which we discuss below.

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 or 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 an expression in a return statement:*
```
return change(E1, E2) % 2;
```
  (d) *Changing a conditional expression:*
```
if (change(E1, E2))
  ... code ...
```
  In the patches that 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:
  (e) *Weakening a condition from A to $A \mid\mid B$:*
```
if (A || change(false, B))
  ... code ...
```

(f) *Weakening a condition from A && B to A*:
```
if (A && change(B, true))
 ... code ...
```
(g) *Strengthening a condition from A || B to A*:
```
if (A || change(B, false))
 ... code ...
```
(h) *Strengthening a condition from A to A && B*:
```
if (A && change(true, B))
 ... code ...
```
We choose a different style of annotations for strengthening a condition from A || B to B and for weakening a condition from A && B to B:

(i) *Strengthening a condition from A || B to B*:
```
if (change(A || B, B))
 ... code ...
```
(j) *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`, then 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 [15] is desired, in our experiments we prioritise divergences that propagate to the output.

(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 [37]. For example:

(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 ...
```
(e) *Weakening the execution of a block of code from conditional on C to unconditional*:
```
if (change(C, true))
 ... code ...
```
(f) *Strengthening the execution of a block of code from unconditional to conditional on C*:
```
if (change(true, C))
 ... code ...
```
Rules 2c, 2d, 2e, and 2f also apply to `while` statements and can be easily tailored to `for` loops.

(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 ...
   ```
   (c) *Replacing code sequences:*
   ```
   if (change(true, false))
    ... old code ...
   else ... new 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.

   However, with those rules developers can easily model large complex patches, spread over not-necessarily contiguous blocks of code. For example, developers might decide to conservatively enclose inside a single `if change(true, false)` statement a large fragment of code that interleaves unmodified and modified bits. While such a heterogeneous code fragment might not necessarily introduce a divergent behaviour, symbolically executing it in the context of the new version would still check for the presence of various kinds of generic errors, including assertion violations. Hence, even approximately outlining the contours of the changed regions in code could prove beneficial.
(4) *Adding/removing variable declarations.* If a variable declaration is added or removed, then 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`.

Automating Patch Annotations. We believe some of the patterns described above could be automated via a compiler pass. However, we acknowledge that the process can sometimes require a good understanding of the semantics of the patch, which go beyond the capabilities of a pattern-based automated technique. Examples include changing a variable type (as in pattern (5) above) or patches with large numbers of inter-dependent parts. In these examples, it would be very challenging to automate the annotation process, except for enclosing the entire patch in a change `(true, false)` statement as noted in pattern (3)(c) above, which is sub-optimal, as explained there.

However, we see Shadow primarily as a development-time tool that would be used by the authors of the code. We found the effort of writing annotations for third-party code to vary from a few minutes for the easy patches up to a few hours for the most complex ones, but the authors of the code would have been able to write them with considerably less effort.

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

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

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 [12, 38]. As the program executes:
   (a) If at a branch point the input exposes a divergence, then we stop concolic 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, then 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 during the concolic phase, we initiate a BSE run in the new version starting from that divergence point (with the symbolic store and path condition that trigger the divergence), 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 [10, 16, 17, 19, 29, 42], shadow symbolic execution can substantially increase memory consumption. As a result, it is important to maximise sharing between the symbolic states 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 the values of 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

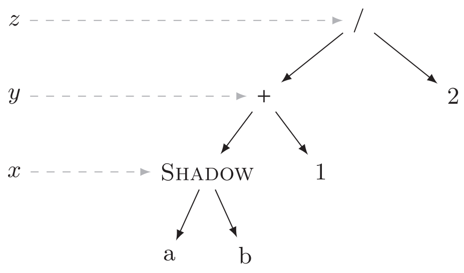


Fig. 6. A shared expression tree for the expressions corresponding to the variables `x, y`, and `z` in Figure 5. Expressions containing shadow subexpressions are kept in the symbolic store and lazily evaluated at symbolic branch points, e.g., in the `if` condition on line 5, to extract the new and old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

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

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 that 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 sanitisation* [39]. The purpose of address sanitisation is to detect heap and stack buffer overflows and use-after-free errors. The technique involves compiler instrumentation of memory allocations, deallocations, and accesses. For instance, stack allocations are instrumented to insert special red zones around the allocated objects, which are later used to detect out-of-bounds accesses. When an error is detected, the execution halts with an error report. Across the experiments that we present further in the article, address sanitisation detected nine unique errors, as identified by the code location and memory address involved in the error.

If the outputs of the two versions differ, then 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 the inputs 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 and uses the concolic execution functionality from the ZESTI extension [26]. Our code is based on KLEE revision 02fa9e4d, LLVM 2.9 and STP revision 1668.

Table 1. An Overview of the Timeout Values Used

Symbolic Run	Total	7,200s
	Test case	3,600s
	Invocation	600s
	BSE	570s
	Solver	30s
Replay	Total	7,200s
	Test case	60s
	Invocation	5s

Invocation means a single run of Shadow or the native version of the program. The BSE timeout is just for symbolic exploration and does not include the time to repeat the concolic phase.

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.[3]

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, the generated inputs should also expose divergences, modulo imprecision in our `change` annotations.

Timeout values for symbolic runs. 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 a maximum of 570s, divided equally among all the divergence points placed in the queue.

For each phase, we set a global timeout of 3,600s for running an entire test case (potentially consisting of multiple invocations). Also, for each phase we set a global timeout of 7,200s for running all the test cases that touch the patch. In all the presented experiments, we use an SMT solver timeout of 30s.

The symbolic run timeout values are summarised in the upper part of Table 1. In Section 5.6, we present an extended study in which we run the experiments with different timeout values to better understand the impact of the time budget on the results.

Timeout values for replay. To rerun the generated inputs natively, Shadow provides a *replay* functionality that implements the enhanced checks described in Section 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 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

[3]We use `gcov`, https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

timeout of 7,200s. Due to some non-determinism in our replay infrastructure, we retry each replay experiment once if the first attempt is unsuccessful. The replay timeouts are summarised in the lower part of Table 1.

Non-determinism. One of the sources of non-determinism are background tasks used in some of the test cases. These test cases would run two instances of a given tool in parallel, one of which would be running in the background. In our infrastructure, we run and replay test cases in a selective manner, each time focusing on the *n*th invocation of the tool. However, it might happen that the *n*th invocation during the symbolic execution run is not the *n*th invocation during the replay run.

To address this problem, we remember the actual invocation parameters and try to match them during the replay phase (rather than relying on the invocation number). However, even this approach is sometimes inadequate as some of the parameters might change at each invocation, e.g., names of temporary files/directories or process IDs.

Finally, Shadow may not generate exactly the same output as the native version of the tool. For instance, a timeout may occur that would prevent the application from completing execution, or some feature might be unsupported by Shadow, e.g., accepting standard input from a Linux pipe. As a result, some test cases might not behave as expected as sometimes the output of a tool is used further on in the test case script. While these are not fundamental limitations, engineering effort would be required to make Shadow more transparent. However, that proved unnecessary for our patch testing experiments.

5 EVALUATION

We evaluate Shadow on the software patches from the `GNU Coreutils` application suite[4] included in the `CoREBench` suite of regression bugs.[5] `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.[6]

The `CoREBench` patches represent a tough challenge for test input generation: as the `CoREBench` authors discuss [5], 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.[7]

`CoREBench` provides 22 pairs of *{bug-introducing, bug-fixing}* patches for `Coreutils`. However, there are only 18 unique *bug-introducing* patches in `CoREBench`, as some patches introduce multiple bugs. These 18 patches are shown in Table 2. 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.

The first column of Table 2 shows the `CoREBench` ID for the patch. If the patch is responsible for multiple bugs, then we show all relevant IDs: e.g., “5 = 16” means that the bug-introducing patch with ID 5 is 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.

[4]http://www.gnu.org/software/coreutils/.
[5]http://www.comp.nus.edu.sg/~release/corebench/.
[6]Measured with the `cloc` tool, http://cloc.sourceforge.net/.
[7]http://sir.unl.edu/portal/index.php.

Table 2. `Coreutils` Patches from `CoREBench`

ID	Tool	Patch Size LOC	Hunks	Files	Test Files Touching	Annotations	Annotations Touched
1	`mv`, `rm`	45	17	4	243	12	12
2	`od`	141	46	1	–	32	–
3	`cut`	294	34	1	17	14	14
4	`tail`	21	4	1	4	4	2
5=16	`tail`	275	12	1	2	1	1
6	`cut`	8	3	1	15	3	3
7	`seq`	148	5	1	29	5	5
8	`seq`	37	4	1	29	12	9
9=18=20	`seq`	324	45	1	–	11	–
10	`cp`	16	8	5	42	2	2
11	`cut`	2	1	1	14	1	1
12=17	`cut`	110	16	1	1	4	4
13	`ls`	13	2	1	8	2	2
14	`ls`	15	5	1	7	4	4
15	`du`	3	1	1	26	1	1
19	`seq`	40	9	1	11	6	6
21	`cut`	31	9	1	11	6	6
22	`expr`	54	6	1	2	4	4

We report the `CoREBench` bug/patch ID, the affected tool, the patch size (which takes into account source code files only), the number of test files that touch the patch, the number of `change()` annotations that we used for each patch, and the number of annotations covered by the test suite.

The *hunks* forming a patch essentially characterise the different non-adjacent segments of code affected by a patch and therefore the number of hunks is indicative of how localised or dispersed the code changes are. More formally, the concept of adjacency is parametric on the so-called context size and a hunk groups together all the lines added or modified in a patch that are at a distance smaller than or equal to the context size. We used the unified diff format with a context size of zero when computing the hunks, which implies that in our setting a hunk is a maximal code fragment consisting of only contiguous lines of changed code. For example, the code in Figure 7 on page 19 consists of three hunks starting on lines 1, 4, and 10. Similarly, the code in Figure 8 on page 20 consists of five hunks starting on lines 2, 8, 11, 14, and 16—note that the changes on the consecutive lines 2 and 3 are enclosed in a single hunk.

As can be seen, the size of a patch varies between only 2 LOC and a single hunk and 324 LOC and 45 hunks. Most patches change a single file, with two exceptions, where 4 and, respectively, 5 different files are affected.

Finally, the last three columns provide information on patch coverage and annotations. The column *Test Files Touching* provides the number of test files in the regression test suite that touch the patch. Those are used as starting points by Shadow and vary between 1 and 243 test cases. The columns *Annotations* and *Annotations Touched* report, respectively, the number of `change()` annotations that we introduced for each patch and the actual number of annotations that are covered by the test suite. We needed to introduce between 1 and 32 annotations across all tested patches. As we can see, in most of the cases all annotations were touched by the test suite, except for patches #4 and #8 for which some of the annotations remained uncovered by the existing tests.

Table 3. Distribution of Annotation Patterns Across the CoREBench Patches

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 second, third, and fourth columns refer to the annotation pattern groups outlined in Section 3.1. The last two columns indicate the presence of changes that require no explicit annotations despite taking some effort to reason about, e.g., an added/removed variable or a type change. 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.

5.1 Annotations

Column *Annotations* in Table 2 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 Section 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 above 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 3 provides a rough overview of the distribution of annotation patterns as classified in Section 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 or an addition and a removal of a variable. The most common annotations were the ones that represent a modification of an rvalue expression. We introduced 80 annotations of that type. We also introduced 27 annotations representing adding or removing extra assignments or conditionals and 17 annotations representing adding or removing straightline code fragments. Furthermore, in 10 out of 18 patches, we encountered added/removed variable declarations or type modifications.

In general, there is often more than one way to annotate a patch and there is a judgement call involved as to what would be the best way to express the change. Furthermore, our manual effort is error-prone, although we are confident that the annotations are correct. Based on our empirical experience with annotating the patches, we created the set of rules outlined in Section 3.1. These rules can be used as a guideline when annotating new patches.

We make our annotations publicly available at http://srg.doc.ic.ac.uk/projects/shadow/annotations.html, 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 2.5GHz 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 2,000MiB 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 that were described in Section 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 in the regression suite that 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 the 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 4 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. As we can see, the BSE phase discovered most of the code-level divergences, orders of magnitude more than the regression test suite and the concolic phase. The concolic phase discovered relatively few divergences on top of the ones found by the test suite. In total, Shadow detected between 1 and 54,842 code-level divergences across all tested patches. However, the figures under *Regression Suite–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 4. 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

	Regression Suite				**Concolic Phase**				**BSE Phase**				**Total**		
	Divergences		**Differences**		**Divergences**		**Differences**		**Divergences**		**Differences**		**Divergences**	**Differences**	
ID	**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	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

The inputs created by Shadow for patch #21, marked with *, also expose two additional bugs (an abort in the concolic and BSE phases and a buffer overflow in the BSE phase), which are different from the one detected by the regression suite (wrong exit code).

Table 5. 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

The second column provides the test inputs generated by Shadow, the third and the fourth columns present the externally visible differences observed between the old and the new version. The last column provides a classification of the externally-visible differences into expected or bugs.

Table 4 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, which 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, then we report it as exposing an expected difference. As we can see, running the regression suite phase led to the detection of differences in 10 out of 16 patches. The concolic phase was able to detect a difference in patch #21 on top of what was found by the regression test suite. The BSE phase found further differences in four patches, notably in patches #6 and #12 for which the other phases had no success. Code-level divergences do not always translate into externally visible differences, as it is the case for e.g., patches #6 and #11 for the concolic phase and #3 and #8 for the BSE phase.

As discussed in Section 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 that fix 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, then 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 non-determinism or 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 5 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

```
 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);
```

Fig. 7. Shadow annotations for `CoREBench` bug #6.

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 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 annotations for the patch are 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).

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 5 shows a test input generated by Shadow that exposes this bug. The input was found in the BSE phase.

In addition to detecting bugs introduced by software patches, Shadow can be used as a debugging tool, to help trace down the root cause of a bug. In particular, we could map the generated test input to the branch point where the divergence was exposed, which could help with debugging. Furthermore, one could track the divergent branch conditions to the `change()` expressions from which they derive. For instance, the bug-triggering input that is shown in Table 5 for the code in Figure 7 was synthesised due to the divergence on line 1, and the divergent branch conditions were due to the `change()` annotation on the same line.

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

Fig. 8. Shadow annotations for CoREBench bug #21.

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

Our annotations for the patch are given in Figure 8. We added six annotations: three of them when an rvalue expression is changed (lines 2, 3, and 14), one that adds a new `if` statement (line 8), one that removes an `if` statement (line 16), and one that modifies an `if` statement (line 11).

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 9 and 14. However, Shadow generated further inputs for which the output differences do not involve error messages. The last two rows of Table 5 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.

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 differences or bugs. Regarding expected differences, 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 [5]. 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, and explore their impact in more detail in Section 5.6.

More generally, some of these patches require a precise environmental model (KLEE's model is incomplete, e.g., it lacks the ability to handle symbolic directories), and at least one patch requires support for symbolic floating-point values (which KLEE does not provide). However, one could implement Shadow in other symbolic execution engines, such as in Cloud9 [6], which extends KLEE with a more comprehensive environmental model, or in one of the recent extensions of KLEE for floating point [23].

We also depend on the availability of inputs that reach the patch (see the discussion in Section 6.2).

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 we typically 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 one could investigate clustering and ranking techniques to help developers sift through these divergences.

We provide a further discussion of limitations in Section 6.3. Below, we present details about experiments performed with increased timeout values and a comparison with a vanilla version of KLEE.

5.6 The Impact of Timeout Values

To understand whether the unsuccessful cases discussed in Section 5.5 were a consequence of timeout values that are too short, we performed further Shadow experiments with increased timeout values.

For this large number of additional experiments, we used several virtual machines running in the cloud and equipped with an 8 core CPU @ 3GHz and 8GB of RAM. Because the hardware platform differs from our initial tests, we first run Shadow with the same timeout values as before, to get baseline results for the cloud. These results are presented in Table 6. Note that for these experiments we report results for patches #5/#16 and #12/#17 separately, as it was easy to test each revision independently, given the parallel platform. As can be seen, the numbers do not deviate much from those in Table 4, except for generating many more bug-exposing inputs in the BSE phase for patch #21.

We next perform a set of experiments with the timeout values changed as described in Table 7. The format and the meaning of the values presented in Table 7 are the same as the ones in Table 1 (see Section 4). For clarity, in *Experiments 1–6* of Table 7, we only present the values that changed compared to the *Baseline* experiment (which are identical with the values from Table 1). Experiments 1–6 present a combination of different timeout values. In experiments 1, 3, 5, and 6, we increased the timeout for a single invocation of Shadow and the timeout for the BSE phase. Furthermore, in experiments 2–6, we increased the global timeout for each phase (denoted as *Total*) from 2h up to 48h for both run and replay. The timeouts for the SMT solver, test case replay and a single tool invocation while replaying remained unchanged for all experiments.

Table 8 presents the total number of divergences as well as the numbers of expected and unexpected differences for each of the experiments. The meaning of the columns is the same as in Table 4; please refer to Section 5.3 for more details.

Discussion. As we can see in Table 8, increasing the timeout values improves the overall results. In most cases the numbers of expected and unexpected differences increased. New differences

Table 6. Baseline for Experiments in the Cloud

	Regression Suite				Concolic Phase				BSE Phase				Total		
	Divergences		Differences		Divergences		Differences		Divergences		Differences		Divergences	Differences	
ID	Total	Replayed	Expected	Bug	Total	Replayed	Expected	Bug	Total	Replayed	Expected	Bug	Total	Expected	Bug
1	2,497	31%	2	-	-	-	-	-	41,082	23%	-	-	43,579	2	-
3	524	100%	-	-	-	-	-	-	16,740	8%	-	-	17,264	-	-
4	6	100%	6	-	-	-	-	-	33	100%	30	-	39	36	-
5	7	100%	-	5	-	-	-	-	7	100%	-	-	14	-	5
16	7	100%	2	3	-	-	-	-	7	100%	-	-	14	2	3
6	232	100%	-	-	15	100%	-	-	1,195	55%	-	81	1,442	-	81
7	62	100%	5	-	-	-	-	-	62	100%	-	-	124	5	-
8	24	100%	-	-	18	100%	-	-	65,733	4%	-	-	65,775	-	-
10	3	100%	-	2	-	-	-	-	3	100%	-	-	6	-	2
11	51	100%	9	-	6	100%	-	-	1,960	87%	-	-	2,017	9	-
12	163	100%	-	-	-	-	-	-	4,497	71%	-	84	4,660	-	84
17	163	100%	-	-	-	-	-	-	4,535	72%	-	99	4,698	-	99
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%	-	-	39,750	26%	-	-	40,052	7	-
21	114	100%	12	4	348	100%	136	122	24,240	12%	4	2,919	24,702	152	3,041
22	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

The timeout values are as in the original study. The meaning of the columns is the same as for Table 4.

Table 7. Timeout Values Used for Additional Experiments

		Baseline	**Experiment 1**	**Experiment 2**	**Experiment 3**	**Experiment 4**	**Experiment 5**	**Experiment 6**
Symbolic Run	Total	2h		8h	8h	48h	48h	48h
	Test case	1h		4h	4h	4h	4h	48h
	Invocation	600s	1,800s		1,800s		1,800s	1h
	BSE	570s	1,740s		1,740s		1,740s	1h
	Solver	30s						
Replay	Total	2h		8h	8h	48h	48h	48h
	Test case	60s						
	Invocation	5s						

For clarity, only the changed values are shown.

were also found: expected differences for patch #3 in experiments 2 and 4, for patch #12/#17 in experiments 3–6, and for patch #19 in experiments 3–6 (in addition to the differences already known for #19). In experiment 4, Shadow detected new bug-revealing differences for revision #8, which uncover a problem not related to the bug reported in `CoREBench`.

It is interesting to observe how changing various timeout values might influence the results. In particular, increasing the timeout for the BSE phase alone does not necessary result in finding more output differences. If we have a test case with three invocations of a tool of interest and a time budget of 1h for running tests, and if we increase the BSE time budget to half an hour, then we might never reach the third invocation of the tool, while that uncovered invocation can be the one that would result in output differences.

5.7 Comparison with Vanilla `KLEE`

To further evaluate the efficiency of Shadow for patch testing, we run experiments using a vanilla version of `KLEE`, borrowing the replay stage with the enhanced checks on both the old and the new version of the code.

We used `KLEE` revision `02fa9e4d`, on which Shadow is based, and ran it on the new versions associated with each patch in `CoREBench`. As a guideline for setting up our experiments, we used the set of options recommended for the `Coreutils` experiments.[8]

We performed two sets of experiments. The first vanilla `KLEE` experiment uses the same timeout values as the baseline Shadow experiment in the cloud, while the second vanilla `KLEE` experiment uses the same timeout values as Shadow experiment 2 in the cloud. The vanilla `KLEE` experiments are also run in the cloud, so the two pairs of experiments are directly comparable.

Vanilla `KLEE` generated between 434 and 1,095 test cases across all revisions in the first experiment and between 838 and 4,374 test cases in the second experiment. The number of differences generated by vanilla `KLEE` are presented in Table 9, with the data for the corresponding Shadow experiments repeated for convenience. As in Table 8, we present the number of differences found by Shadow across all stages, including running the test suite with enhanced cross-version checks.

The experiments indicate that Shadow discovers more divergences and differences as compared to vanilla `KLEE`. For the first set of experiments (with baseline timeout values), Shadow detected expected or unexpected output differences in 14 out of 18 revisions, while vanilla `KLEE` was successful only in 5 out of 18 revisions. Similarly, in the second set of experiments (with experiment 2 timeout values), Shadow was able to find output differences in 15 out of 18 revisions, while `KLEE` was successful in 5 revisions. In particular, Shadow detected an order of magnitude more

[8]http://klee.github.io/docs/coreutils-experiments/.

Table 8. Additional Experiments for Different Timeout Values

	Baseline			Experiment 1			Experiment 2			Experiment 3			Experiment 4			Experiment 5			Experiment 6		
	Total	Differences		Total	Differences		Total	Differences		Total	Differences		Total	Differences		Total	Differences		Total	Differences	
ID	divs	Exp.	Bug	divs	Exp.	Bug	divs	Exp.	Bug	divs	Exp.	Bug	divs	Exp.	Bug	divs	Exp.	Bug	divs	Exp.	Bug
1	43,579	2	-	33,191	2	-	182,708	2	-	144,795	2	-	361,314	2	-	381,513	2	-	373,950	2	-
3	17,264	-	-	14,641	-	-	85,037	5	-	34,279	-	-	84,700	6	-	32,414	-	-	14,672	-	-
4	39	36	-	39	36	-	39	36	-	39	36	-	40	36	-	39	36	-	39	36	-
5	14	-	5	14	-	5	14	-	5	14	-	5	14	-	5	14	-	5	14	-	5
16	14	2	3	14	2	3	14	2	3	14	2	3	14	2	3	14	2	3	14	2	3
6	1,442	-	81	2,245	-	59	4,716	-	260	4,756	-	274	4,709	-	261	4,345	-	263	2,595	-	164
7	124	5	-	124	5	-	124	5	-	124	5	-	124	5	-	124	5	-	124	5	-
8	65,775	-	-	62,606	-	-	72,738	-	-	91,656	-	-	80,377	-	74	25,001	-	-	55,179	-	-
10	6	-	2	4	-	2	6	-	2	6	-	2	6	-	2	6	-	2	6	-	2
11	2,017	9	-	1,845	9	-	3,393	9	-	3,571	9	-	3,392	9	-	3,568	9	-	3,635	9	-
12	4,660	-	84	6,363	-	78	8,418	-	374	11,731	-	464	8,425	-	374	11,464	370	64	10,110	-	417
17	4,698	-	99	6,369	-	78	8,436	-	376	11,775	67	396	8,429	55	324	11,085	66	370	10,135	49	366
13	11	1	1	11	1	1	11	1	1	11	1	1	11	1	1	11	1	1	11	1	1
14	2	-	-	2	-	-	2	-	-	2	-	-	2	-	-	2	-	-	2	-	-
15	1	1	-	1	1	-	1	1	-	1	1	-	1	1	-	1	1	-	1	1	-
19	40,052	7	-	37,069	7	-	100,514	7	-	90,988	8	-	102,088	30	-	99,068	18	-	91,628	20	-
21	24,702	152	3,041	18,783	152	3,150	73,248	160	11,587	60,703	162	10,220	58,277	1,022	23,683	66,421	788	26,113	53,067	686	26,644
22	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 9. Comparison Between Shadow and Vanilla KLEE in Terms of the Number of Expected and Unexpected (Bug) Differences

	Shadow baseline		**Vanilla KLEE run 1**		**Shadow experiment 2**		**Vanilla KLEE run 2**	
ID	**Expected**	**Bug**	**Expected**	**Bug**	**Expected**	**Bug**	**Expected**	**Bug**
1	2	-	-	-	2	-	-	-
3	-	-	-	-	5	-	-	-
4	36	-	22	-	36	-	31	-
5	-	5	54	9	-	5	234	56
16	2	3	69	1	2	3	288	-
6	-	81	-	-	-	260	-	-
7	5	-	-	2	5	-	-	-
8	-	-	-	-	-	-	-	-
10	-	2	-	-	-	2	-	-
11	9	-	-	-	9	-	-	1
12	-	84	-	-	-	374	-	-
17	-	99	-	-	-	376	-	-
13	1	1	-	-	1	1	-	-
14	-	-	-	-	-	-	-	-
15	1	-	-	-	1	-	-	-
19	7	-	-	-	7	-	-	-
21	152	3,041	105	308	160	11,587	423	1,079
22	-	-	-	-	-	-	-	-

The two approaches are given the same time budget.

unexpected differences for revision #21 as compared to vanilla KLEE. These results are not surprising, given that Shadow's objective is to find divergences, while KLEE has a generic symbolic execution approach. However, vanilla KLEE manages to find expected and, respectively, unexpected differences in patches #5 and #7, which Shadow does not, and there are also cases where vanilla KLEE generates more inputs exposing externally-visible differences. This is because unlike Shadow, which starts from a fixed test suite (the regression test suite in our experiments), vanilla KLEE has the freedom to explore other parts of the program space. One possible improvement would be to first run vanilla KLEE and add the test cases it generates to the starting test suite for Shadow.

In general, vanilla KLEE has the advantage of not requiring a test suite; however, the user needs to specify the inputs to the program that are to be made symbolic (in our experiments, the number and the sizes of the command-line arguments and files), which Shadow does not require.

Finally, we note that the results for vanilla KLEE are also much more non-deterministic across runs due to the internal search heuristics used by KLEE. For instance, while the longer vanilla KLEE experiment finds more differences across the board, it misses the bugs in patches #7 and #16, which the shorter vanilla KLEE experiment detects.

5.8 On the Complexity of SMT Solver Queries

In all our experiments, we used an SMT solver timeout of 30s (as noted in Table 1) and our empirical evaluation shows that the solver timeout was not the bottleneck.

We measured the number of queries and the number of query constructs—where a query construct is a node of a symbolic expression such as a constant or a binary operator—for both the Shadow and the vanilla KLEE experiments. The results are summarised in Table 10.

Table 10. The Maximum, Average, and Median Number of SMT Queries and Query Constructs in the Shadow and Vanilla KLEE Experiments

	Queries		Query constructs	
	Shadow	**Vanilla**	**Shadow**	**Vanilla**
Maximum	39,487	14,016	271,695	918
Average	138	8,704	624	373
Median	10	9,220	50	391

The figures for the Shadow experiments include both the concolic and BSE phases.

Table 10 shows that the maximum number of queries across Shadow experiments was higher than for vanilla KLEE. However, both the average and the median values suggest that Shadow issues much fewer queries than vanilla KLEE. Similarly, the table shows that the maximum and the average number of query constructs we recorded was significantly higher for Shadow than for vanilla KLEE. However, the median value for Shadow is almost eight times lower, which suggests there were more small-sized queries in Shadow as compared to vanilla KLEE.

In summary, the results presented in Table 10 suggest that the queries solved in Shadow are overall both less numerous and more lightweight when compared to vanilla KLEE. However, there were also cases in which Shadow had significantly more and/or significantly more heavyweight queries than vanilla KLEE.

6 DISCUSSION

We structure our discussion section into reflections on the regression testing process (Section 6.1) and Shadow's reliance on concrete inputs (Section 6.2), limitations of the Shadow approach (Section 6.3), and applications of Shadow to other programming languages (Section 6.4) and software engineering problems (Section 6.5).

6.1 Reflections on the 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.

6.2 Reliance on Concrete Inputs

We designed Shadow to rely on concrete inputs that reach the patch. The alternative would have been to have Shadow also try to synthesise inputs that reach the patch. While this would have given Shadow more freedom to explore the search space of the program, we opted for assuming the existence of concrete inputs for two reasons. First, the challenges of generating inputs that reach the patch and that of further generating inputs that trigger divergences across versions are quite distinct, and we did not see a benefit from addressing them in the same system. Second, the challenge of patch reachability has already been addressed in the past [1, 24, 25, 27, 35, 40, 43], and Shadow can reuse these results. In fact, Shadow can use any approach for obtaining such inputs: regression test suites (as done in this work), vanilla symbolic execution (as suggested in Section 5.7), directed symbolic execution (as in `KATCH` [27]), or directed greybox fuzzing (as in AFLGo [4]).

6.3 Limitations

We further discuss some of the main limitations of Shadow.

Incompleteness of symbolic execution. Symbolic execution is incomplete in that it usually only explores a subset of all feasible execution paths in a given time budget. Furthermore, it may also miss feasible paths when interacting with the external environment, such as the operating system, or when SMT queries time out. For Shadow, this often means that only a subset of the divergences introduced by a patch are discovered in a given time budget.

The impact of the initial set of inputs. The actual inputs that touch the patch can have a significant influence on the efficiency of the testing process. These inputs determine the control-flow path followed by symbolic execution. In particular, certain values may prevent the analysis from reaching parts of the patch. For example, consider a fragment of code containing a branch point `if (x > 4)`, with `x` a symbolic input. When executing such code starting with `x = 10`, the branch condition is added to the constraint set for the execution path and it effectively limits the set of possible values for `x` further down on the path, even if the patch can be reached with a value of x smaller than or equal to 4.

Multi-hunk patches. Real-world patches are likely to consist of multiple hunks, i.e., multiple disjoint areas of the code are likely to get modified within a single commit to the source code repository. Indeed, as we show in Table 2, all but two of the `CoREBench` patches consist of more than one hunk. For such multi-hunk patches, we consider that a test input touches the patch if it executes at least one line of code in at least one of the hunks. A limitation of this test case selection strategy is the possibility of missing a certain class of bugs that are caused by inter-dependencies between hunks, i.e., that are triggered when hunks are executed in a certain sequence. If our selected inputs touch just a single hunk, then we might miss such bugs. While the BSE stage helps in some cases, it might happen that for some multi-hunk changes we will start the BSE run on the first encountered change and never reach the code of the subsequent hunks due to the large number of executed paths. As a result, the bugs that depend on executing a sequence of changed code fragments might be missed by Shadow.

Data-only divergences. Shadow focuses on the control-flow divergences introduced by the patch code. Although shadow expressions can be created at any program location in the code (via our `change` annotations), Shadow will only consider divergences once it encounters a branch point that involves a shadow expression in the branch condition. This design decision can result in missing a bug that is triggered by a changed state of a program rather than a changed control flow. An example of such a bug would be a modification of a value that is then printed out to the screen, but that does not affect the control flow. Shadow could be extended to handle data divergences by specifying points of interest: in the example above, this would be the printing function.

Technical limitations. Shadow builds on top of the `KLEE` symbolic execution engine. As a result, it inherits all its technical limitations. In particular, `KLEE` operates at the level of `LLVM` bitcode and can only reason symbolically about the `LLVM` bitcode; external calls to native code are possible, but the symbolic data needs to be concretised first. Therefore, for fully functional symbolic analysis, the whole analysed program and its external dependencies should be first compiled into bitcode. Furthermore, while `KLEE` symbolically executes `LLVM` bitcode, it mainly supports bitcode generated from C programs. For instance, support for high-level programming language concepts like C++ lambdas depends on whether and how they are translated into the LLVM bitcode by the compiler. It is important to note, however, that these limitations are not fundamental and could be solved with engineering effort.

6.4 Shadow Symbolic Execution for Other Programming Languages

Even though Shadow is built on top of KLEE and focuses on C programs, the technique is independent of the programming language in which the code is written: the same principles of annotating patches and reducing the search space for symbolic execution could be applied to other programming languages.

A different research group has implemented the shadow symbolic execution approach for Java programs [30] on top of the Symbolic PathFinder (SPF) framework [33]. Results indicate that the technique is also effective for performing regression unit testing of Java code—it reduced the number of test inputs generated compared to vanilla SPF (which also generated tests irrelevant to regression testing) and synthesised regression test inputs that were missed by SPF.

6.5 Applications of Shadow Beyond Patch Testing

While our objective for shadow symbolic execution is to expose divergences introduced by patches that are written by humans, the technique is also useful in other software engineering approaches that require inputs exposing divergences between either manually written or automatically generated program variants.

For instance, shadow symbolic execution is used by Chekam et al. [9] in the context of investigating the "clean program assumption" and determining the correlation between fault revelation and various forms of coverage, where the availability of inputs that trigger the differences between faulty and fixed versions is critical.

Another application of shadow symbolic execution is discussed by the authors of N-Prog [18], a framework that combines bug detection with test suite augmentation. N-Prog generates multiple variants of a program such that each of the variants passes all tests in the original test suite, and then it runs the original program and the variants on a random input stream. If for a given input stream the outcome is different in at least one of the program variants, then it means that either the variant is incorrect or that a bug/interesting new test case was detected. The availability of inputs that expose divergences between these versions is essential to the successful application of the technique.

Shadow symbolic execution was also discussed recently in the context of genetic improvement [41]. Genetic improvement tasks such as automatic program repair or automatic specialisation could benefit from the ability of shadow symbolic execution to generate inputs that trigger differences between the automatically-generated variants, with differences defined appropriately. Those inputs can be used to validate the results (e.g., whether the generated patch fixes the bug) and/or to provide guidance to the genetic improvement process.

7 RELATED WORK

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, 24, 25, 27, 35, 40, 43]. 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 [34] is a general framework that can reason about program differences, but its reliance on summaries raises significant scalability issues.

Directed incremental symbolic execution [35] 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 that 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.

A technique with a similar goal to Shadow is presented by Qi et al. [36]. The idea is to use dynamic symbolic execution to generate test cases that exercise the patch code and also have externally visible effects that propagate to the output. The approach uses the control flow graph and the control dependency graph to guide symbolic execution toward the changed code, and also the path constraints associated with the old and the new version. Shadow requires that at least one of the test inputs touches the patch and also needs the manual patch annotation step, which the discussed approach does not. Unlike Shadow, the approach does not run the two versions in the same symbolic execution instance and does not run full symbolic execution on the divergent paths. Also, while Shadow can analyse whole patches, the discussed approach can only process one changed statement at a time.

Another similar approach for testing software changes is Conc-iSE [13]. The goals of Shadow and Conc-iSE are similar, but the main focus of Conc-iSE is on concurrent patches. Conc-iSE automatically infers the execution paths affected by the patch using change impact analysis and symbolic summaries and executes the affected paths incrementally. To obtain the summaries, Conc-iSE needs to first symbolically execute the old version of the program. Shadow executes both versions of the program at the same time and relies on existing test inputs, without the need to perform a symbolic run up front.

SymDiff [21] is a static analysis technique for finding bugs in software patches. The high-level idea is to verify whether a set of assertions that hold for the old version also hold for the new version of a program. The relative strengths of Shadow and SymDiff are related to the relative advantages of static and dynamic analysis: for instance, SymDiff can verify bug fixes but can also generate false alarms, while Shadow can only reason about code that it executes and thus miss divergent program paths, but has no false alarms and can generate actual test inputs that trigger the divergences it explores.

The techniques discussed above were evaluated on patches significantly less complex than the `Coreutils` patches we considered. However, Shadow is not fully automatic; while many of the annotations that we added could be automated, manual assistance might still be needed. Nevertheless, research on automating this step is promising [22, 32]; furthermore, note that even an imprecise automatic annotation system might be enough to help Shadow 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 [29, 42], model checking [10], product line testing [19], and software updating [16].

Research on test suite augmentation requirements has used the differences between two program versions to derive requirements that test suites have to meet to ensure proper patch testing [14, 37]; our analysis could potentially provide further information to guide these techniques.

8 CONCLUSION

In this article, 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.

ACKNOWLEDGMENTS

We thank Paul Marinescu and Daniel Liew for their assistance during this project, and Frank Busse, Alastair Donaldson, Sophia Drossopoulou, Antonio Filieri, Timotej Kapus, Andrea Mattavelli, Martin Nowack, Alessandro Orso, and the anonymous reviewers for their feedback on the work.

REFERENCES

[1] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically directed dynamic automated test generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'11)*.
[2] Marcel Böhme, Bruno C. D. S. Oliveira, and Abhik Roychoudhury. 2013. Partition-based regression verification. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*.
[3] Marcel Böhme, Bruno C. D. S. Oliveira, and Abhik Roychoudhury. 2013. Regression tests to expose change interaction errors. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*.
[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*.
[5] Marcel Böhme and Abhik Roychoudhury. 2014. Corebench: Studying complexity of regression errors. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*.
[6] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11)*.
[7] Cristian Cadar and Hristina Palikareva. 2014. Shadow symbolic execution for better testing of evolving software. In *Proceedings of the 36th International Conference on Software Engineering, New Ideas and Emerging Results (ICSE NIER'14)*.
[8] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: Three decades later. *Commun. Assoc. Comput. Mach.* 56, 2 (2013), 82–90.
[9] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*.
[10] Marcelo d'Amorim, Steven Lauterburg, and Darko Marinov. 2007. Delta execution for efficient state-space exploration of object-oriented programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'07)*.
[11] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: Introduction and applications. *Commun. Assoc. Comput. Mach.* 54, 9 (Sept. 2011), 69–77.
[12] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the Conference on Programing Language Design and Implementation (PLDI'05)*.
[13] Shengjian Guo, Markus Kusano, and Chao Wang. 2016. Conc-iSE: Incremental symbolic execution of concurrent software. In *Proceedings of the 31th IEEE International Conference on Automated Software Engineering (ASE'16)*.
[14] Rajiv Gupta, Mary Jean Harrold, and Mary Lou Soffa. 1996. Program slicing-based regression testing techniques. *Softw. Test. Verificat. Reliabil.* 6 (1996), 83–112.

[15] Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. 2001. *A Practical Tutorial on Modified Condition/Decision Coverage*. Technical Report NASA/TM-2001-210876. NASA.

[16] Petr Hosek and Cristian Cadar. 2013. Safe software updates via multi-version execution. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*.

[17] Petr Hosek and Cristian Cadar. 2015. Varan the unbelievable: An efficient N-version execution framework. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*.

[18] Martin Kellogg, Benjamin Floyd, Stephanie Forrest, and Westley Weimer. 2016. Combining bug detection and test case generation. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE'16)*.

[19] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. 2012. Shared execution for efficiently testing product lines. In *Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE'12)*.

[20] Miryung Kim and David Notkin. 2006. Program element matching for multi-version program analyses. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR'06)*.

[21] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential assertion checking. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*.

[22] Wei Le and Shannon D. Pattison. 2014. Patch verification via multiversion interprocedural control flow graphs. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*.

[23] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair Donaldson, Rafael Zähl, and Klaus Wehrle. 2017. Floating-point symbolic execution: A case study in N-version programming. In *Proceedings of the 32nd IEEE International Conference on Automated Software Engineering (ASE'17)*.

[24] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2011. Directed symbolic execution. In *Proceedings of the 18th International Static Analysis Symposium (SAS'11)*.

[25] Paul Dan Marinescu and Cristian Cadar. 2012. High-coverage symbolic patch testing. In *Proceedings of the 19th International SPIN Workshop on Model Checking of Software (SPIN'12)*.

[26] Paul Dan Marinescu and Cristian Cadar. 2012. make test-zesti: A symbolic execution solution for improving regression testing. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*.

[27] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage testing of software patches. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*.

[28] Paul Dan Marinescu, Petr Hosek, and Cristian Cadar. 2014. Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*.

[29] Matthew Maurer and David Brumley. 2012. TACHYON: Tandem execution for efficient live patch testing. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security'12)*.

[30] Yannic Noller, Hoang Lam Nguyen, Minxing Tang, and Timo Kehrer. 2018. Shadow symbolic execution with Java PathFinder. *ACM SIGSOFT Softw. Eng. Notes* 42, 4 (2018), 1–5.

[31] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a doubt: Testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*.

[32] Nimrod Partush and Eran Yahav. 2014. Abstract semantic differencing via speculative correlation. In *Proceedings of the 29th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'14)*.

[33] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: Integrating symbolic execution with model checking for java bytecode analysis. *Auto. Softw. Eng.* 20, 3 (Sep 2013), 391–425.

[34] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. 2008. Differential symbolic execution. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE'08)*.

[35] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. In *Proceedings of the Conference on Programing Language Design and Implementation (PLDI'11)*.

[36] Dawei Qi, Abhik Roychoudhury, and Zhenkai Liang. 2010. Test generation to expose changes in evolving programs. In *Proceedings of the 25th IEEE International Conference on Automated Software Engineering (ASE'10)*.

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

[38] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*.

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*.

[40] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2011. eXpress: Guided path exploration for efficient regression test generation. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'11)*.

[41] The 50th CREST Open Workshop—Genetic Improvement 2017. *Retrieved from* http://crest.cs.ucl.ac.uk/cow/50/.

[42] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. 2009. Efficient online validation with delta execution. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*.

[43] Zhihong Xu and Gregg Rothermel. 2009. Directed test suite augmentation. In *Proceedings of the 16th Asia-Pacific Software Engineering Conference (ASPEC'09)*.

Received November 2017; revised May 2018; accepted June 2018