Regression Tests to Expose Change Interaction Errors

Marcel Böhme
School of Computing
National University of Singapore
marcel@comp.nus.edu.sg

Bruno C.d.S. Oliveira
School of Computing
National University of Singapore
oliveira@comp.nus.edu.sg

Abhik Roychoudhury
School of Computing
National University of Singapore
abhik@comp.nus.edu.sg

ABSTRACT

Changes often introduce program errors, and hence recent software testing literature has focused on generating tests which stress changes. In this paper, we argue that changes cannot be treated as isolated program artifacts which are stressed via testing. Instead, it is the complex dependency across multiple changes which introduce subtle errors. Furthermore, the complex dependence structures, that need to be exercised to expose such errors, ensure that they remain undiscovered even in well tested and deployed software. We motivate our work based on empirical evidence from a well tested and stable project - Linux GNU Coreutils - where we found that one third of the regressions take more than two (2) years to be fixed, and that two thirds of such long-standing regressions are introduced due to change interactions for the utilities we investigated.

To combat *change interaction errors*, we first define a notion of change interaction where several program changes are found to affect the result of a program statement via program dependencies. Based on this notion, we propose a *change sequence graph* (CSG) to summarize the control-flow and dependencies across changes. The CSG is then used as a guide during program path exploration via *symbolic execution* - thereby efficiently producing test cases which witness change interaction errors. Our experimental infrastructure was deployed on various utilities of GNU Coreutils, which have been distributed with Linux for almost twenty years. Apart from finding five (5) previously unknown errors in the utilities, we found that only one in five generated test cases exercises a sequence that is critical to exposing a change-interaction error, while being an order of magnitude more likely to expose an error. On the other hand, stressing changes in isolation only exposed half of the change interaction errors. These results demonstrate the importance and difficulty of change dependence aware regression testing.

General Terms

Maintenance and Evolution

FSE '13 St. Petersburg, Russia

1. INTRODUCTION

Changes even to well-tested software projects can introduce subtle bugs of varying severity that may be exposed only years later. Such change-based errors in deployed software come in two forms. First of all, bug fixes may introduce new bugs. For instance, Gu et al. [6] mentions that feature additions or bug fixes, introduce new bugs in 9% of cases. Secondly, a subtle or poorly understood "interaction" among various changes may introduce hard-to-find errors in well-tested code, which then get deployed. In this paper, we focus on test generation to expose such subtle *change interaction errors* (CIEs).

Evidence of subtle change interaction errors can be found in many well-tested and deployed software projects. In our study on `GNU Coreutils`, we found that *every fifth bug fix actually patches regressions* introduced in an earlier commit. About one third of these regressions take more than two (2) years to find and fix, despite the tool set being rather well tested. Note that ∼21% of the total commits update the comprehensive test suite, while only 30% actually update the utilities (the remaining 49% are related to maintenance, like documentation, the build process, or ambiguous error messages). Thus, it is surprising that on utilities with such well-updated test-suites, errors due to change interaction will remain for two years. In fact, the `GNU Coreutils` have been dispatched with almost every Linux distribution for the last 20 years!! This led us to think that change interaction errors, which stress subtle dependencies across changes, may be hard-to-find due to most regression testing methods being focused on some form of *coverage*.

At this point, we step back and review the recent regression testing research which focus on program changes. A recent work [16] presents criteria and experiments for the interaction among program changes but does not suggest any method for integrating them into regression testing. Among the works achieving change aware test generation, some study only independent program changes [1, 12]. Several of the testing methods attempt to achieve either a structural coverage of changed statements or some other structural coverage (such as branch outcome coverage) in the modified program (e.g., see [21]). Since coverage based methods may not stress the semantic effect of the changes, attempts have been made to take a powerful symbolic execution based path exploration engine, and adapt it to the presence of program changes. Since symbolic execution captures the semantic effect of program changes, the hope is that the semantic effect of a change can be propagated through such methods. On the other hand, since a

full-fledged symbolic execution based path exploration can be exceedingly slow, these methods employ various pruning strategies to cull away program paths which cannot reach or propagate changes (e.g., see [17]). Other authors suggest to statically compute the program slices for every change and dynamically employ symbolic execution upon these slices to exercise all paths that are affected by a changed statement (e.g., see [11]). However, in all of these works, the set of changes in a program is treated in an aggregate fashion. The flows/dependencies across changes are not systematically explored/exploited for generating test cases.

In this paper, we present a test generation method to systematically explore and expose subtle errors arising due to the "interaction" among program changes. Since any such change interaction leading to errors is inherently dynamic, we first statically approximate the relationships among the changes. Our approximation is called *change sequence graph* (CSG) which captures (i) the control-flow across the changed statements and (ii) the control-flow to control locations at which multiple changes may interact, leading to unexpected semantic effects. These interaction locations are computed based on the program dependencies across multiple changed statements. The CSG is then used as a definitive guide to find out the sequence of control locations that need to be visited for exposing potential change interaction errors. These control locations are visited systematically by programming a graph-based search strategy on top of the directed symbolic execution engine, Otter [8].

Experimental results from our approach on `Coreutils` show the prevalence of change interaction errors among regression bugs. We note that the `GNU Coreutils` tool-set is a collection of Linux utilities which have been widely tested. In particular, every fifth commit to the repository updates a comprehensive test suite that exists for more than twenty years, and the tool set was further tested by the authors of `klee` [3] and `test-zesti` [9] (reporting 3 and 2 errors, respectively). Despite such extensive testing, we found and reported five verified, previously unknown regression errors, apart from many known errors. Among other notable findings, we noticed that two in three differential errors can be classified as change interaction errors. We also found that only half of the CIEs were exposed by a testing algorithm that target changes in isolation, but does not account for their interaction. This clearly demonstrates the importance of change-interaction aware regression testing.

In summary, the **contributions** of this paper are:

- **Change-Interaction Errors:** We identify and formalize change-interaction errors: errors that happen in evolving software, which arize due to the combined semantic impact of multiple changes. We argue for the importance of this class of errors with a study of regression on GNU coreutils over a period of 5 years.
- **Detection Method for CIEs based on CSGs:** We propose a datastructure called a change-sequence graph to capture potential sequences of changed statements and interaction locations in an execution of a program. Using CSGs we show a detection method which stresses sequences in the graph to expose CIEs.
- **Implementation and Empirical Evaluation:** Our CIEs detection method has been implemented and an empirical evaluation using that implementation was conducted to evaluate its effectiveness.

2. REGRESSION IN `GNU Coreutils`

To study software regression, we looked at the repository of `GNU Coreutils`, which has been actively developed and maintained for more than twenty years. Our results show that within the last five years every fifth bug fix actually patches regressions introduced earlier and that 30% of such regressions take more than 2 years to be fixed. These results are corroborated by the package maintainer.

2.1 Statistics of Regression

It is possible to access the history of every change commited to the source code repository of `GNU Coreutils` since Oct'92[1]. Usually, these commits are accompanied by a commit message that describes the relevance and intention of the change. The commits to the repository of `GNU Coreutils` are categorized as changes to particular tools, or as *build*, *tests*, *maint*[enance], amongst others. The developers adopted this commit message labeling about five years ago. This allows us to distinguish code-changing commits[2] from maintenance commits. Parsing the commit messages for keywords, such as "bug", "fix", or "regression", we were able to find how many of the code-changing commits are bug fixes and feature additions. If the commit message contained "introduced" or "regression", we could derive whether a bug fix was actually patching regressions introduced earlier. Often, a *regression-fixing* commit would reference the *regression-introducing* commit. Thus, we can measure the time in-between. As the commits have been nicely categorized in the last five years, we looked at those between Jan'08 and Feb'13. However, *regression-fixing* commits can reference *regression-introducing* commits that were submitted much earlier. Given the X- and the (logarithmic) Y-axis in Figure 1, the graph shows that X percent of the *regression-introducing* commits 1) require *more than Y days* to be found and fixed (solid line), and 2) contain *more than Y Changed Lines of Code* (CLoC; dashed line).

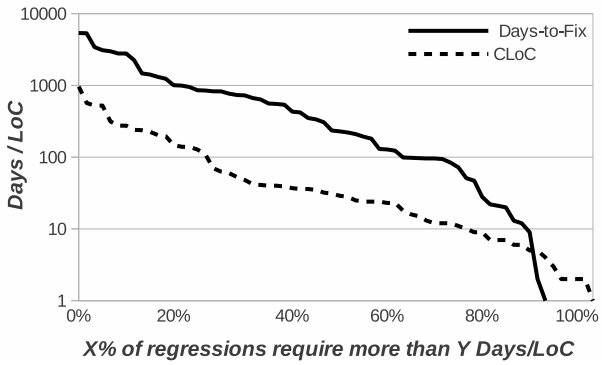

Figure 1: Regression Statistics - `GNU Coreutils`

Results. About 30% of the 2.6k commits in the recent 5 years are code changes - feature additions and bug fixes in roughly equal shares. Interestingly, *every fifth bug fix actually patches regressions* introduced in an earlier commit. The following observations are are further corroborated by package co-maintainer, Pádraig Brady, via email-exchange:

[1] `http://git.savannah.gnu.org/cgit/coreutils.git`
[2] Code-changing commits are labeled by the changed tool.

O.i 30% of the regressions introduced in earlier commits take more than 2 years to find and fix despite a comprehensive and well-maintained test suite (∼21% of the total commits update the test suite)

O.ii 45% of the regressions are introduced when more than 35 LoC are changed (while only about 25% of the code-changing commits modify 35 LoC or more).

This led us to suspect that the changed behaviour introduced by the syntactic changes to the tools is not properly tested. In particular, we consider the subtle interplay of many code changes as reasons for regressions to be exposed so late. We call this type of errors — *change-interaction errors*. Indeed, as discussed in Section 7, we find that 66% of the errors introduced in earlier commits can only be exposed by input exercising certain *critical sequences* of changed statements. A critical sequence turns out to be exercised half as likely as a non-critical sequence, while being 15 times more likely to expose an error. In the remainder of this section, we have a closer look at one of the regression errors.

2.2 Buffer Overflow in `cut`

During our investigations, we found and reported a buffer overflow in the tool `cut` of `GNU Coreutils` which was introduced as a regression error in commit `ec48bead` and manifests as SEG_FAULT for the failing test input.[3] Buffer overflows can be exploited maliciously to gain root access to affected computers [4]. This issue is particularly critical for systems that are dispatched with almost every Linux distribution, such as `GNU Coreutils`, which contains well-known command-line tools, such as `cp`, `mv`, `echo`, and `cut`. Fortunately, in the five years preceeding this paper the package maintainer of `GNU Coreutils` had to fix only 10 SEG_FAULTs.[4] However, a surprising 6 out of 10 are regression errors *introduced in earlier commits*.

2.2.1 The Anatomy of a Regression

In simple terms, the tool `cut` takes a set of number ranges, a file, and an optional output-delimiter as input and prints the content of every line in the specified file within the specified ranges, optionally separated by the specified output-delimiter. For instance, the command in Figure 2, uses "hello world" as input to the `cut` utility - which prints the range between the 2nd and 3rd character, and from the 7th character onwards, both ranges separated by "`,`" (comma).

```
$ echo "hello world" | cut -output-del=, -b2-3,7-
el,world
$
```

Figure 2: Linux Terminal - the output of `cut`

Problem. *If there are no finite ranges (e.g., `7-`), then too much memory is unnecessarily allocated.*
Specifically, if `max_range_endpoint` is set in line 504 of Figure 3 or earlier, then the array `printable_field` is allocated `max_range_endpoint` of memory (line 509). If `output_delimiter_specified`, then `printable_field` is unnecessarily (but successfully) accessed at `eol_range_start` in line 534. Note, if `eol_range_start>max_range_endpoint`, then `max_range_endpoint` is set to `eol_range_start` in line 504.

Intended Change. *Memory allocation only if necessary.* Specifically, only if `max_range_endpoint` is set, allocate the array `printable_field` with `max_range_endpoint` of memory. Only if `output_delimiter_specified` and `max_range_endpoint` is set, then the array `printable_field` shall be accessible in line 534.

```
265  : bool is_printable_field (size_t i)
266  :    return printable_field[i];
 ..  :
503--: if (max_range_endpoint < eol_range_start)
504--:   max_range_endpoint = eol_range_start
 ..  :
508++: if (max_range_endpoint)
509  :   printable_field = malloc(max_range_endpoint+1)
 ..  :
531  : if (output_delimiter_specified
532  :     && !complement
533  :     && eol_range_start
534++:     && max_range_endpoint
     :     && !is_printable_field (eol_range_start))
535  :   mark_range_start (eol_range_start)
```

Figure 3: SEG_FAULT introduced in `cut`

Actual Changes. *The developer applies three code changes. Every change is essential to fix the memory leak.*
Specifically, the developer *C.1)* adds that `printable_field` is allocated only if `max_range_endpoint` is set (line 508), *C.2)* adds that `printable_field` is accessed only if `max_range_endpoint` is set (line 534), and *C.3)* removes that `max_range_endpoint` is set to `eol_range_start` if `eol_range_start > max_range_endpoint` (lines 503-504). Note, all changes are essential to fix the memory leak. For instance, without change *C.3*, the variable guarding the memory allocation is always set, rendering the additional checks of changes *C.1* and *C.2* redundant.

Regression Error. *If finite ranges are specified, then unallocated memory can be accessed, yielding a SEG_FAULT.*
Specifically, if 1) `max_range_endpoint` is set, 2) `max_range_endpoint < eol_range_start`, and 3) `output_delimiter_specified` is set, then the array `printable_field` is accessed out-of-bounds at `eol_range_start` in line 266.

2.2.2 Combined Semantic Impact of Changes

The observation of the regression error depends on the execution of both changes, *C.1* and *C.2*. They have a combined "semantic impact" on the same program location - the memory access. Specifically, the allocation of memory for `printable_field` in line 509 depends on the code added with change *C.1*. The access of memory in `printable_field` in line 266 depends on the code added with change *C.2*. Because the *success* of accessing an array also depends on the memory allocation for this array, both changes have a combined impact at the memory access location. So, the memory access at line 266 is called *interaction location* of *C.1* and *C.2*. The sequences in which the changes can be executed are depicted in Figure 4.[5] Note, *C.3* is not part of the presented graphs since a deletion does not manifest in the changed version P'.

[3]Report and fix avail. at `http://debbugs.gnu.org/13627`.
[4]We analysed commit messages in the source repository. The actual number may be greater.
[5]For brevity, we removed sequences that contain a change but no memory allocation or access.

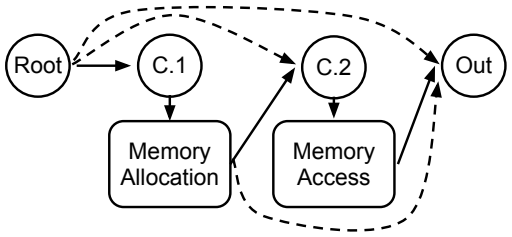

Figure 4: Input can exercise these change sequences.

It is insufficient to test both changes in isolation. The regression error is only observable for (some) input that exercises both changes - the sequence following the solid lines in Figure 4. The buffer overflow is not observable for input exercising only change *C.1* but not *C.2* and neither for input exercising only change *C.2* but not *C.1* - sequences exercising one of the dashed lines in Figure 4. Hence, we call this regression a *change interaction error*.

3. ERRORS IN SOFTWARE EVOLUTION

This section formally describes classes of errors that can occur during software evolution. In particular, we are interested in a class of errors, arizing from the interaction of multiple changes, that we call *change interaction errors* (CIE). To establish the context of CIEs, we also define a useful generalization of regression errors, which we call *differential errors*.

3.1 Preliminaries

For the definitions in this section, we will assume two successive versions of a program, P and P', and an oracle S. The oracle S specifies the intended behavior for P and P'. As such, it is expected that for all input i executed on P', the output is observationally equivalent[6] to executing i on the oracle S. Note that we abstract over what the oracle S is: it could be a specification, the ultimate final version of a program, a validating test suite, or some other artifact that could be used to validate expected behaviour. Using earlier version P, the changed version P' and intended behavior S of P and P', we can more formally define what we mean by a *regression error*.

Definition 1 (*Regression Error*)

An error is a regression error if for some input i holds: $P(i) \not\equiv P'(i)$ and $P(i) \equiv S(i)$.

In other words, a regression error happens when for some input i the earlier version, P, works as expected but the new version, P', does not work anymore. Note that this definition *does not* prevent P' from exposing the correct behaviour for some other input, which fails in P w.r.t. S. Therefore, our definition of regression error captures the common situation in which the initial version P may have some errors that are intended to be fixed in P', but while P' is fixed for some inputs, it starts behaving incorrectly for some other inputs. An intended software quality improvement turns into a possible detoriation of the software quality.

[6]Two programs P and P' are observationally equivalent for an input i, $P(i) \equiv P'(i)$, if the *relevant* program output produced by executing i on P and P' is the same.

3.2 Differential Errors

In the context of software evolution we often find the need for a notion more general than that of a *regression error*. We call this notion *differential error*.

Definition 2 (*Differential Error*)

An error is a differential error if for some input i holds: $P(i) \not\equiv P'(i)$ and $P'(i) \not\equiv S(i)$.

In other words, a differential error happens when, for some input i, the changed version P' works differently from both, the earlier version P and the intended behavior S. There are two interesting situations. The situation in which the earlier version $P(i)$ worked as expected ($P(i) \equiv S(i)$) is just equivalent to the definition of regression error. On the other hand, the situation in which the earlier version $P(i)$ did not work as expected either ($P(i) \not\equiv S(i)$) cannot be called regression error. So we call it differential error. This captures a situation, e.g., of an incomplete fix. The developer intends to fix the behaviour of P, so that test cases i and j fail on P w.r.t. S. But while i may now pass in the fixed version P' and j produces different output, j may still fail on P' w.r.t. S – the fix was incomplete. In practice, it is helpful to characterize situations in which several intermediate "fixes" are implemented until an ultimate version meets the expectations.

3.3 Change Interaction Errors

A change-interaction error is a special kind of differential error. Informally, a change-interaction error happens when multiple changes are introduced in a program, and those multiple changes interact in unexpected ways. More formally we can this class of errors as follows.

Definition 3 (*Change interaction error (CIE)*)

A change-interaction error happens when there exists a sequence of changed statements $\vec{C}$, such that both of the following conditions hold:
1) there exists an input i that exercises all changed statements in $\vec{C}$ in order and $S(i) \not\equiv P'(i)$;
2) for every input j that skips the execution of at least one changed statement in the sequence $\vec{C}$, we have that $S(j) \equiv P'(j)$.

We call the sequence $\vec{C}$ the *critical sequence* of the CIE. That is $\vec{C}$ corresponds to a sequence of changed statements that is necessary to expose the error. Any smaller sequence that skips the execution of at least one changed statement in that sequence cannot expose the error.

3.4 Running Example

For illustration purposes, we use the two concrete program versions P and P' in Figure 5 to explain salient concepts in the remainder of this work. The two programs are simplified extracts of two versions of the Linux core utility `cut` - the behavior of which is explained in Section 2.2.1. The code is related to the parsing of the user-provided number ranges for the tool. As long as `*fs` points to a character of the string, it tests whether the character is a digit (line 1), a dash (line 6) or the end of the line (line 9). If the character is a digit, then the number is read into value `v`. In the changed version a boolean `lhs` is set to `true` (lines 4-5). If the character is a dash, the variable `init` is computed using `v` (lines 7-8). If the end of the line is reached, the bug is observable if `init` is 0 (line 10).

P

```
1
2  while(true){
3   if(isDigit(*fs)){
4    v = rdDigit(*fs );
5
6   else if(*fs == '-')
7    c = v;
8    init = (c)? v : 1;
9   } else if (*fs == EOL) {
10   assert(init != 0);
11   break;
12  }
13  fs++;
14 }
```

P'

```
1  bool lhs = false;
2  while(true){
3   if(isDigit(*fs)){
4    v = rdDigit(*fs);
5    lhs = true;
6   } else if (*fs == '-'){
7    c = lhs;
8    init = (c)? v : 1; //IL
9   } else if (*fs == EOL){
10   assert(init != 0);
11   break;
12  }
13  fs++;
14 }
```

Figure 5: Core Utility `cut.v1` changed to `cut.v2`

The *changed statements* are highlighted in grey. There are three changed statements in the changed version, which can be identified using the corresponding line numbers: $\{1, 5, 7\}$. We should point out that our notion of change is syntactic, purely textual and corresponds to code changes that manifest in the *changed version* (P'), such as added or modified statements. In other words, changed statements can be determined using textual differencing tools, like `diff`. The use of `diff` has the advantage that it works for any two programs, although it can be quite imprecise. There are other, more precise ways to deal with changes, but these typically assume some form of alignment between the two program versions [1, 12]. Unfortunately, these alignment assumptions do not always hold for the real programs that we are interested in. For this reason we chose the less precise, but unrestricted approach using `diff`.

Change Interaction Error. In the program P' in Figure 5 a CIE happens when the input string is "0-". In this case the following sequence of changed statements is executed: $\langle 1, 5, 7\rangle$. Before entering the loop, line 1 is executed. Since the first character is '0', the first iteration of the loop meets the condition at line 3 and the changed statement in line 5 is executed. At this point the variable v is set to 0 and the variable `lhs` is set to `true`. In the second iteration of the loop the condition at line 6 is met and the change in line 7 is executed. Since `lhs` is true, `init` is set to 0 (as `v` is 0). In contrast, for the same input, program P sets the variable `init` to 1. Consequently, in the last iteration of the loop, the assertion in line 10 is violated for P', but not for P.

Note that only a specific sequence of changes (as well as a specific input) triggers this error. The interaction of the changed statements in lines 5 and 7 at the statement in line 8 causes this error. The combined semantic impact of both changes lead to the differential evaluation of the conditional expression `(c)?v:1` in both versions, P and P'. Other change sequences, such as $\langle 1, 7, 7\rangle$, $\langle 1, 7, 5\rangle$, $\langle 1, 7, 7, 5\rangle$, will not expose the error.

4. CHANGE SEQUENCE GRAPH

To support detection of change-interaction errors (CIE) we propose a statically computed structure which we call *change-sequence graph* (CSG). A change sequence graph approximates the computation of potential CIEs by using *control-flow information* to derive sequences in which the changed statements can potentially be exercised and *dependence information* to derive locations at which the changes can potentially interact.

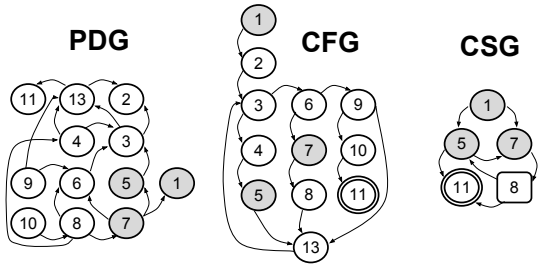

Figure 6: CFG, PDG and CSG for P' in Figure 5.

4.1 Potential Interaction

To aid detecting CIEs, we can approximate all potential sequences of changed statements in a program using control-flow information from a control-flow graph (CFG). Essentially, a potential sequence of changed statements corresponds to a path in the CFG that contains changed statements. Having information about every potential sequences of changed statements is helpful because all critical sequences will be included in those sequences. In other words, this information will allow us to build a detection method for CIEs that searches potential critical sequences and exposes CIEs.

We are particularly interested in change sequences where the changed statements interact. That is, each executed changed statement has some impact on the output, and not executing one of those statements can lead to a different output. It is in this class of sequences where we can find change interaction errors. To detect such sequences, one useful definition is that of a potential interaction location of a sequence of changed statements.

Definition 4 (*Potential Interaction Location*)

A statement s is a potential interaction location of a sequence of changed statements $\vec{C}$, if s (statically) data- or control-depends on more than 1 changed statements in $\vec{C}$.

Information about potential interaction points can be computed using the program dependency graph (PDG). Essentially, we utilize the backward slice of the statement s to compute the set of changed statements that can have a semantic impact on s. If the set contains more than one different changed statement, then s is a potential interaction location of those statements. Note that an interaction location can coincide with a changed statement.

For our running example, Figure 6 depicts the CFG, the PDG, and third structure that is discussed in a moment. The statement at line 8 is a potential interaction location of the changed statements in lines 5 and 7, since it transitively depends on both the changed statements. As such, both changes can have a combined semantic impact on this control-location, effectively causing the regression error.

The notion of potential interaction location allows us to define an approximation of *change interaction*.

Definition 5 (*Potential Change Interaction*)

A sequence of changed statements $\vec{C}$ is potentially interacting if there are potential interaction locations for $\vec{C}$.

If there is no interaction location for two changed statements, since there is no information-flow, they are guaranteed not to interact and can be tested in isolation.

Algorithm 1 Change-Sequence Graph Construction

Require: Programs P and P'
1: let $C_{Code} \leftarrow diff(P, P')$
2: let $CFG \leftarrow markedCFG(P', C_{Code})$
3: let $PDG \leftarrow markedPDG(P', C_{Code})$
4: let $CSG \leftarrow markedNodesOf(CFG)$
5: **for all** Change $c \in CFG$ **do**
6: traverseChange(c, c)
7: **end for**
8:
9: **function** traverseChange($curr$, c)
10: **for each** *node* that directly follows *curr* in CFG **do**
11: **if** *node* is change or output **then**
12: add edge from c to *node* in CSG
13: **else**
14: let $C_I \leftarrow$dependsOnChanges(*node*, PDG)
15: **if** $|C_I| > 1$ **then**
16: add *node* to CSG
17: **for each** $c \in C_I$ **do**
18: add edge from c to *node* in CSG
19: **end for**
20: traverseChange(*node*, *node*)
21: **else**
22: traverseChange(*node*, c)
23: **end if**
24: **end if**
25: **end for**
26: **end function**
Ensure: Change-sequence graph CSG.

The information about all potential sequences of changed statements and potential interaction points can be synthesized in a *change-sequence graph* (CSG). Thus a CSG represented a subset of program paths in a program where change-interaction errors *may* exist. Other program paths, which are not represented in the CSG, cannot have change-interaction errors as they do not contain change sequences.

4.2 Computing the Change Sequence Graph

The CSG can be computed using the CFG and the PDG for the changed program P'. Algorithm 1 shows the detailed construction of the CSG. The inputs of the algorithm are two programs P and P' and the output is the change-sequence graph CSG. The first step is to compute the changed statements between P and P' (line 1). As discussed in Section 3, this can be done using the `diff` tool. The next step is to compute the annotated versions of the CFG and PDG of P' (lines 2−3). Both, the CFG and PDG are annotated with information about the changed statements. Initially the CSG contains no edges, only nodes. These nodes are the changed statements and output nodes that are recovered from the CFG using the procedure $markedNodesOf$ (lines $4-5$). The final step of the algorithm is to iterate through all the changed statements in the CFG and, for each change, use the auxiliary function traverseChange to add the relevant edges and interaction locations to the CSG (lines $5-7$).

The recursive function traverseChange takes two arguments *curr* and c. The first argument represents the current node in the CFG. The second argument represents the changed statement that edges may have to connect to. For each node in the CFG, which directly follows from the current node, we have three possibilities for the *node*:

Change or output node (lines $11-12$): If we reach some other change node, this indicates that there *may* be a control-flow from the change c to this change. Thus, we add a corresponding edge to the CSG to indicate such potential flow. Similarly, if we reach an output node, we should add an edge between change c and that node to indicate the potential control-flow.

Interaction location (lines $14-20$): If the *node* is an interaction location, it is added to the CSG and connected. Specifically, the function dependsOnChanges(*node*, PDG) computes the changed statements that can have a semantic input on *node* using the PDG. If there is more than one change having a semantic impact on *node*, then *node* is an interaction location and is added to the CSG connected to the changes it depends on. Conceptually, every interaction location can be regarded as a new change. To trace the semantic impact of the interaction location, we keep recursively traversing the CFG by invoking traverseChange with both arguments set to *node*. Naively, the function dependsOnChanges can return all changed statements in the static backward slice of *node*. To optimize, it may choose a *node* to be an interaction location only if *node* is an "important" interaction location in some respect. For instance, given an interaction location i for change sequence $\vec{C}$ and a statement s that directly depends on i. While s is also an interaction location of $\vec{C}$, it may not be an important one. Alternatively, the interaction locations could be computed by taking the static forward slice for every changed statement and marking their intersection as interaction location.

Neither of above (line 22): Any other CFG node should be ignored in the CSG. This is achieved by calling traverseChange with *node* as first argument. This effectively sets *curr*, representing the node in the CFG that is currently traversed to *node* and implies *node* will not appear in the CSG.

5. SEARCH-BASED INPUT GENERATION

To expose change interaction errors, and differential errors in general, test cases are generated. The exploration technique uses the Change Sequence Graph as a guide to exercises the structure of inter-dependencies across the changed statements. We employ symbolic execution along these dependencies.

The search-based input generation is depicted in Algorithm 2. The algorithm takes two program versions, P and P', and the CSG (cf. Alg. 1) as input and computes a set of difference-revealing test cases T. We adopted the directed symbolic execution algorithm as discussed by Ma et al. [8]. However, instead of searching for input that exercises any target in a specified (flat) set of targets, we extended the algorithm to search a specified directed graph of targets (i.e., the CSG). The search algorithm is presented independent of the search strategy.

Algorithm 2 is initialized in the first five lines. It starts with an empty test suite T and the first set of changed statements in the CSG (those without incoming edges). These are added as targets for the symbolic state *symbState* which is created in lines 2-4. A symbolic state is essentially an intermediate state of symbolic execution and has three main properties - (i) a statement *next* which is to be executed next, (ii) a partial path condition *pc*, that is satisfied by every input exercising the same program path until s, and (iii) a set of *targets*.

Algorithm 2 Search-Based Input Generation

Require: Programs P and P'; Directed Graph CSG
1: let $T \leftarrow \emptyset$
2: let $symbState.targets \leftarrow CSG.startNodes$
3: let $symbState.pc \leftarrow true$
4: let $symbState.next \leftarrow P'.firstStmt$
5: let $states \leftarrow \{symbState\}$
6: **while** $states \neq \emptyset \wedge \neg isTimeout()$ **do**
7: let $bestState \leftarrow chooseBestState(states)$
8: let $s \leftarrow symbExec\_next(bestState, P')$
9: **if** $isBranch(s)$ **then**
10: let $stateT \leftarrow bestState.pc \wedge s.branchCond$
11: let $stateF \leftarrow bestState.pc \wedge \neg s.branchCond$
12: remove $bestState$ from $states$
13: add $stateT$ and $stateF$ to $states$
14: **else if** $s \in bestState.targets$ **then**
15: **if** s is an output **then**
16: let $t \leftarrow smt\_solve(bestState.pc)$
17: **if** $P(t) \neq P'(t)$ **then**
18: add t to T
19: **end if**
20: remove $bestState$ from $states$
21: **else**
22: $bestState.targets \leftarrow$ next targets of s in CSG
23: **end if**
24: **end if**
25: **end while**
Ensure: Difference-revealing test cases T.

The symbolic execution of a symbolic state can be *resumed* at any time and *pauses* if a branch or a target is reached. The first symbolic state *symbState* is created with $pc = true$, statement *next* is set program start, and the *targets* are assigned to the first set of changed statements in line 4. In the following line 5, it is added to the empty list of symbolic *states*.

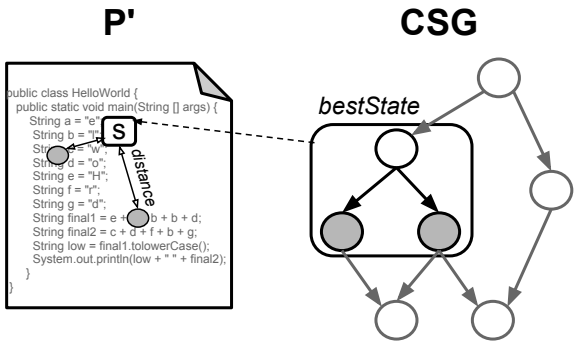

Figure 7: The *bestState* is choosen with the shortest distance in the source code of P' from s to the target (left). Once a target is reached, the symbolic state moves to the target's children in the CSG (right).

The search commences in line 6. As long as the list of *states* is non-empty and no timeout occurs, the search works as follows. From the list of *states* the *bestState* is chosen according to a given search strategy, which is implemented in *chooseBestState*. For instance, as depicted in Figure 5, every symbolic state is assigned a measure of distance to its targets, ranked according to this measure, and chosen if it has the shortest distance. We further prioritize states with a greater proportion of targets that are yet unreached by other symbolic states. In line 8, the *bestState* resumes the symbolic execution of P' until s, the next statement to be executed, becomes either a branch or one of the targets to be reached. If s is a branch (lines 9-12), then two states are created - one following the *true*-branch and the other following the $false$ branch. The path conditions and the list of *states* are updated accordingly. If s is a *target* (lines 13-23), then we further distinguish whether or not s is an output statement. If s is an output statement, then we solve the path condition using a Satisfiability Modulo Theory solver to derive a concrete program input t (line 15). This input is executed on both versions to validate whether t exposes a behavioral difference. If so, t is added to the set of difference-revealing test cases T. Since *bestState* reached the output, it requires no further symbolic execution and can be removed from the list of *states* (line 19). Otherwise, if s is a target of *bestState* and not an output statement (line 21), then we set as new *targets* of *bestState* the nodes following the outgoing edges of the reached node in the CSG. The right-hand side of Figure 5 shows the *bestState* searching for two CSG nodes (in grey). If *bestState* finds the node on the left, the next target of *bestState* becomes that bottom left node.

6. EMPIRICAL EVALUATION

6.1 Implementation and Setup

We have implemented Algorithm 2 into the directed symbolic execution tool, Otter [8]. The user provides two versions of a C program compiled into the C Intermediate Language (CIL) and a text file with a representation of the CSG.

Otter provides a wide choice of search strategies which implement the function *chooseBestState* in Algorithm 2. For our experiments we used one of the most efficient[7] strategies. The best symbolic state is chosen based on the shortest distance to the targets computed in the interprocedural control-flow graph. Occasionally, the next state is chosen randomly. We extended the search strategy by prioritizing states with a greater proportion of yet unreached targets. Instead of searching for a global set of targets, our implementation extends a symbolic state to have its own set of targets. Once a target is reached, the children of the reached target become the new targets for this state. The execution of a symbolic state terminates only if the output has been reached and thus no more further targets are to be reached. We then compute a concrete input satisfying the path condition of a state that reached an output node. This input is executed on both program versions. The information from the standard unix pipes `stdout` and `stderr` describes the program output. If the output differs in more than the program name, the test case and its differing output is reported. If the user *optionally* provides a "golden version", our implementation can classify the observed differential output further as "differential error" (i.e., regression/incomplete fix) or "progression".

We executed our implementation on a desktop computer with an Intel®Core™2 Quad CPU at 2.83GHz and 4GB of main memory to generate test cases within the time frame of 5 minutes. The same sequence of changed statements can be exercised by multiple generated test cases.

[7] `RoundRobin(RandomPath,InterSDSE-efficient)`.

6.2 Subjects

We chose the subjects according to the following criteria:

Known Regressions. For every regression, we know i) the earlier version, ii) the regression-introducing version, iii) the bug report(s), and iv) the regressing-fixing version(s). The analysis of known regressions increases the credibility of the subjects and reduces the scope of non-maintenance commits which we need to inspect.

Multiple Changes. In this study, we are not interested in the semantic impact of single changed statements but the interplay of multiple changed statements. Therefore, we consider only regressions involving multiple changed statements.

Deterministic Behavior. The execution of the same input on the same program always yields the same output. Determinism is a prerequesite for every testing technique and as such also for symbolic execution.

Version Pair	Fixed in Revision	Commit Date	Bug Report @ http://lists.gnu.org/
seq.v0→seq.v1	seq.v2	09.07.2007	2007-07/msg00055.html
16.06.05→01.07.06	seq.v3	14.02.2009	2009-02/msg00139.html
	seq.v3	14.02.2009	2009-02/msg00139.html
seq.v1 →seq.v2	**seq.v4**	24.11.2012	2012-11/msg00145.html
01.07.06→09.07.07	**seq.v5**	10.01.2013	2013-01/msg00054.html
cut.v0→cut.v1	cut.v4	07.02.2011	2011-02/msg00036.html
02.06.04→04.12.04	**cut.v6**	24.11.2012	2012-11/msg00151.html
cut.v1→cut.v2	cut.v3	22.05.2007	2007-05/msg00195.html
04.12.04→22.05.07	**cut.v5**	18.11.2012	2012-11/msg00114.html
cut.v6→cut.v7	**cut.v8**	05.02.2013	2013-02/msg00011.html
24.11.12→06.12.12			
expr.v0→expr.v1	expr.v2	26.05.2005	2005-05/msg00189.html
16.11.04→14.01.05			

Table 1: Subjects - Version history

We study six version pairs that together introduced 11 regression bugs, five of which are found and reported by our method (in **bold** font). Table 1 shows the considered `Version Pairs` that the latter version of which introduces bugs that are `Fixed in` the a subsequent revision. The fixes are presented with `Commit Date` and `Bug Report`. The bug being fixed with `cut.v4` is further discussed by Marinescu and Cadar [9] and together with `cut.v9` only observable as buffer overflow. We inserted an assertion that states that an array shall never be accessed out of bounds. The tools `cut`, `seq`, and `expr` consist of about 900, 500, and 900 Lines of Code (LoC), respectively. However, these tools utilize monolithic, shared libraries, prompting colleagues to quote between 2k to 3k effective LoC for the smallest tools [3] up to 20k instructions for the largest tool [9] in `GNU Coreutils`.

6.3 Research Questions

During the empirical evaluation of the change-interaction guided regression test generation technique, we want to answer the following research questions.

RQ.1 Severity. How many differential errors can be classified as change interaction errors? What is the probability to exercise a sequence exposing a change interaction error compared to other sequences?

RQ.2 Efficacy. How many differential errors are exposed by a test generation technique that exercises changes in isolation as compared to one that considers their inter-dependencies and change interactions?

7. RESULTS AND ANALYSIS

7.1 Result Presentation

Table 2 shows the bugs introduced when changing the given versions, whether these are change interaction errors and the test cases generated by our CSG-guided test generation technique. The first two columns show the errors introduced by the changes of the `Version Pairs` that are `Fixed in` the versions given in the second column. For instance, when *seq.v1* was changed to *seq.v2*, errors are introduced that are fixed in versions *seq.v3*, *seq.v4*, and *seq.v4*. Errors highlighted in **bold** face were previously unknown and subsequently reported by us. The subsequent four columns show the results for the generation of test cases exercising the `Change Sequence Graph`, while the latter three columns show the results for a test generation technique that considers sufficient to exercise every changed statement, effectively treating `Changes in Isolation`. Both groups of columns have a similar format. Column `#Tests` depicts the number of test cases generated. Column `#Diff` depicts the number of test cases revealing a difference when executed on both versions. Some of the semantic differences are expected (progression). Column `#Error` depicts the number of test cases that are not expected and expose the respective error. An error, that is exposed only by input exercising a sequence of changed statements but not by input "skipping" statements in that sequence, is classified as change interaction error (Col. `CIE`).

Version Pairs	Sequence	%Test	%Error
seq.v0 → seq.v1	non-critical	19.02%	0.00%
	critical	80.98%	1.39%
seq.v1 → seq.v2	non-critical	99.50%	0.30%
	critical	0.50%	100.00%
cut.v0 → cut.v1	non-critical	96.83%	4.09%
	critical	3.17%	100.00%
cut.v1 → cut.v2	non-critical	87.40%	11.71%
	critical	12.60%	33.33%
cut.v6 → cut.v7	non-critical	95.68%	0.00%
	critical	4.32%	28.57%
expr.v0 → expr.v1	non-critical	71.43%	0.00%
	critical	28.57%	16.67%

Table 3: `%Test` generated test cases exercise a (non-) critical sequence. `%Error` generated test cases exercising a (non-) critical sequence expose an error.

Table 3 shows the percentage of tests exercising critical sequences versus the percentage of tests exercising non-critical sequences. One test case exercises exactly one sequence. A *critical sequence* is a sequence of changed statements that is relevant to expose a change interaction error. The first column depicts the `Version pairs` considered, followed by whether the results refer to `critical` or `non-critical` sequences. The latter two columns are explained by example of the last row: "On average, one quarter of the generated test cases for the version pair expr.v0 and expr.v1 exercise a critical sequence. From those, every sixth exposes an error".

To generate the test suites that stress changes in isolation (see RQ.2), we generated test cases that cover every changed statement that is also exercised by the approach presented in this paper. We set as targets the output and such statements that have the greatest depth in the chain of control-dependencies. In other words, instead of a graph of targets, we provided a set of targets. Otherwise, we employed the same tool and search strategy.

Version Pair	Fixed in	RQ1: Change Sequence Graph CIE	#Tests	#Diff	#Error	RQ2: Changes in Isolation #Tests	#Diff	#Error
seq.v0 → seq.v1	seq.v2	x	163	43	6	205	65	0
	seq.v3	x	163	43	5	205	65	0
seq.v1 → seq.v2	seq.v3	-	200	26	2	200	21	17
	seq.v4	-	200	26	3	200	21	0
	seq.v5	x	200	26	1	200	21	0
cut.v0 → cut.v1	cut.v4	-	379	42	30	471	42	30
	cut.v6	x	379	42	12	471	42	12
cut.v1 → cut.v2	cut.v3	-	254	228	162	453	201	58
	cut.v5	x	254	228	26	453	201	5
cut.v7 → cut.v8	cut.v9	x	324	4	4	342	6	6
expr.v0 → expr.v1	expr.v2	x	42	2	2	82	2	2
Average (per version pair)		*7/11*	*227*	*57.5*	*25.3*	*292.2*	*55.8*	*21.7*

Table 2: Bugs introduced in version pairs, fixed in later versions, are witnessed by the generated test cases.

RQ.1 Change Interaction Errors

Two third of the differential errors can be classified as change interaction errors. Only one in five test cases exercise a critical sequence, being 15 time more likely to expose an error.

Using our implementation, we have found and reported four of the seven listed change-interaction errors and one more differential error, that were previously unknown. On average, 227 test cases were generated that *exercise* a change sequence (see Table 2). Every fourth test case *propagates* the combined semantic effect of the exercised changed statements to the output and thus makes a difference observable. While many of these expose expected behavioral changes, every tenth test case exposes a differential error.

Change interaction errors are *subtle*. On average, only 21.7% of the generated test cases exercise a critical sequence (see Table 3). On the other hand, the malicious *effect* of a critical change sequence is much greater than that of a non-critical sequence. Only 3.2% of the test cases exercising a non-critical sequence expose an error versus 50% exercising a critical sequence. Test cases exercising a critical sequence are 15.6 times more likely to expose an error than test cases exercising a non-critical sequence. That suggests that the changes in these critical sequences are interacting in a negative and unintended form.

RQ.2 Comparison to Changes in Isolation

Only 57% of the change interaction errors are exposed by test cases generated to stress changes in isolation.

To compare, we generated a test suite that covers every changed statement which is also covered by the test suite generated using a change sequence graph. On average, 292 test cases were generated that *exercise* a change sequence (cf. Table 2). Every fifth test case *propagates* the combined semantic effect of the exercised changed statements to the output and thus makes a difference observable. Many of these expose expected behavioral changes, every 15th test case exposes a differential error – significantly less than our CSG-based test generation approach. Within five minutes, using our CSG-based approach every error is witnessed by 25 test cases on average. In contrast, using the other approach that considers changes in isolation only seven of the eleven errors are witnessed by, on average, 18 test cases each. In particular, only 57% of the change interaction errors are exposed by test cases generated to stress changes in isolation as compared to 100% by our technique.

8. THREATS TO VALIDITY

The main threat to *external validity* is the generalization of the results. During our study of `GNU Coreutils` we encountered several regression errors that can only be observed when certain environmental conditions are satisfied. One example is an error that was reported to occur specifically on a Solaris 32-bit maching and could not reproduced on other machines. Depending on the program environment, the same test case may or may not expose an error. In fact, the package co-maintainer of `GNU Coreutils`, Pádraig Brady, noted in an email correspondence that it may be unclear even for the experienced developer, exactly how to write the test cases in the presence of such non-determinism. He suggested to introduce an explicit interface for file operations. This suggests a lack of modelling the environment [13], or concurrency [19] during the testing process. As discussed in Section 6.2, our experimental subjects and regression errors are chosen so that the observability of an error does not depend on the program environment but on source code properties. The conclusions should be viewed in the same context.

The main threats to *internal validity* are T.1) the search strategy that was utilized and T.2) the practical absence of assertions that mark an error within symbolic execution. T.1) The experimental results depend on the utilized search strategy. A less efficient search strategy may have exposed less differential errors within the same amount of time. However, the utilized search strategy does not prioritze critical over non-critical sequences. Thus, it does not affect the main conclusion of *RQ.1*. We utilized the same search strategy for the experiments that compares to testing changes in isolation. Thus, it does not affect the main conclusion of *RQ.2*. T.2) Symbolic execution requires highlighting of error states, for instance, by assertions. In Section 3.4 and Table 1, we list the versions `cut.v3` and `cut.v5` as bug fixes for regressions introduced in an earlier version of `cut`. The regressions are observable as buffer overflows. However, without the explicit assertion stating that an array should never be accessed at an index greater than its size, the symbolic index for this array may often concretize as small number, such as 1 or 0, but never as a number that has more than the nine digits necessary to witness these particular overflows. While our *implementation* is able to find error-exposing test cases in the presence of such assertions, it is unable to find error-exposing test cases in their absence.

9. RELATED WORK

Test Suite Augmentation aims at generating new test cases that stress the changed behaviour in a program. Typically, this is done by exploiting *knowledge about changes* and using *symbolic execution* techniques - which are also key ideas in our approach. However, the main novelty of our work is the consideration of the inter-dependencies among multiple changes during test generation. Our technique effectively exercises sequences of changed statements and potential interaction locations. Existing techniques either discuss the semantic impact of single changes only [1, 12], or do not systematically consider the interaction and inter-dependencies among multiple code changes [14, 17].

Test Suite Augmentation techniques can be distinguished in *semantic approaches* [2, 10], that are based on the program summaries of both versions to compute the semantic changes, and *syntactic approaches*, that are directed by the syntactic changes to exercise paths that may expose semantic changes. The syntactic techniques can be further distinguished into those seeking to re-establish code coverage of a test suite after the program is changed [21], those following the Reach-Infect-Propagate[8] approach [1, 14, 12], and those exercising every program path affected by a change [17, 11].

Techniques, such as eXpress [17] or DiSE [11], that exercise every program path *affected* by changed statements, are finer-grained and less scalable than our approach. The focus on affected code regions makes these techniques more efficient than full path exploration approaches, like DART [5], since less paths are to be explored. However, these techniques may still exercise *many different paths within the same sequence* of changed statements; paths that may or may not contain interaction locations; paths that may all expose the same error. More systematically, our CSG directed TSA approach targets *sequences and interaction locations of changed statements* instead of all *affected paths*. In practice this means that once a difference revealing test case is found for a sequence, unexplored affected paths that can still realize this sequence do not no have to be explored further.

TSA techniques based on Reach-Infect-Propagate (RIP) [14, 12, 15] follow a motivation similar to our work: Instead of exploring *every path* affected by changes, the RIP approaches deem it sufficient to find *one path* that executes a change, infects the program state, and propagates to the output. However, existing techniques consider the semantic effects of the changes in isolation. For the subjects in our experiments, a technique based on this consideration could expose only half of the change interaction errors. In the presence of multiple changes, the approach of Santelices et al. [14] requires a change-free path from the change to the program start - effectively a change in isolation.

Coverage-based TSA techniques seek to re-establish code coverage when the program is changed [21, 20]. However, to expose change-interaction errors and understand the combined semantic impact of multiple changes, it is insufficient to merely exercise every change, as discussed earlier.

Semantic TSA techniques [2, 10] require the computation of a differential semantic program summary for both versions to determine the semantic changes. While this approach is sound and very precise, it may be less scalable.

[8]Reach a change, infect the program state, and propagate the infection to the output [18].

Change Interaction. Santelices et al. [16] propose a formal definition of change interaction: two changes c_1 and c_2 interact in an execution if removing one of the changes alters the semantic effect of the other change on that execution. This notion of change interaction is too precise. For our practical purposes, detecting such changes interactions cannot be done in an efficient manner. Essentially, given a test case t and code changes C that are applied to program P yielding P', there are $2^{|C|}$ program configurations to be analyzed, each with only a subset of C applied to P. Our definition of potential change interaction approximates the above definition and can be computed more efficiently. A set of changed statements C potentially interacts if there exists a statement that syntactically depends on every $c \in C$.

Reachability. In order to explore change sequences our approach builds and extends previous work that deals with reaching statements in a program. However, these tools seek to reach a single statement [12, 22], a set of statements [8], or a sequence of statements [7] instead of a graph. To overcome this problem we have modified the Otter tool [8] to take a graph of statements as input and target multiple statements along this graph structure at once.

10. CONCLUSION

When a program evolves due to feature additions, bug fixes, or other code quality improvements, the source code of the program is changed. Especially when multiple developers change the code base at different places, comprehending the semantic impact of such code changes and potential interactions is difficult. Consequently, errors that result from a poor understanding of the inter-dependencies across changes are often introduced in the code base.

In this paper, we have argued for the importance and subtleness of such change interaction errors, which are pervasive even in well-tested and widely used software. Since existing regression testing techniques do not adequately stress code where change interaction may occur, we have proposed a new regression testing technique that addresses these limitations. Our recipe for exposing change interaction errors employs a judicious mix of flows, dependencies and semantic effects across changes. In other words, to witness a change interaction error — multiple changes should be executed (flow information), multiple changes should affect a potential interaction location via data- and control dependencies (dependence information), and the semantic effect of a change should not get masked. In our approach, the control flow between changes is captured in the Change Sequence Graph, dependencies across changes are witnessed in potential interaction locations, and we attempt to exercise these dependencies and propagate their semantic effects via symbolic execution on the changed program. Our experiments on Linux `GNU Coreutils` demonstrate the effectiveness of such an approach in hunting down hard-to-find change interaction errors even in well-tested software projects.

11. ACKNOWLEDGMENTS

We are grateful for the engaging discussions with the package (co-)maintainers of the `GNU Coreutils`, Jim Meyering and Pádraig Brady. Both shared their valueable insights in the testing of a real, widely distributed project that is being successfully maintained for over 20 years. This work was partially supported by Singapore's Ministry of Education research grant MOE2010-T2-2-073.

12. REFERENCES

[1] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*, TAIC-PART '06, 2006.

[2] M. Böhme, B. C. Oliveira, and A. Roychoudhury. Partition-based regression verification. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE 2013, pages 301–310, 2013.

[3] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*, pages 209–224, 2008.

[4] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security '98*, pages 346–355, 1998.

[5] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, 2005.

[6] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *ICSE '10*, pages 55–64, 2010.

[7] W. Jin and A. Orso. Bugredux: reproducing field failures for in-house debugging. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, 2012.

[8] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *Proceedings of the 18th international conference on Static analysis*, SAS'11, 2011.

[9] P. D. Marinescu and C. Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *ICSE'12*, pages 716–726, 2012.

[10] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA, 2008. ACM.

[11] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, 2011.

[12] D. Qi, A. Roychoudhury, and Z. Liang. Test generation to expose changes in evolving programs. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, 2010.

[13] D. Qi, W. N. Sumner, F. Qin, M. Zheng, X. Zhang, and A. Roychoudhury. Modeling software execution environment. *Reverse Engineering, Working Conference on*, 0:415–424, 2012.

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

[15] R. Santelices and M. J. Harrold. Applying aggressive propagation-based strategies for testing changes. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 11–20, Washington, DC, USA, 2011. IEEE Computer Society.

[16] R. Santelices, M. J. Harrold, and A. Orso. Precisely detecting runtime change interactions for evolving software. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, 2010.

[17] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. express: guided path exploration for efficient regression test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, 2011.

[18] J. M. Voas. Pie: A dynamic failure-based technique. *IEEE Trans. Softw. Eng.*, 18(8):717–727, Aug. 1992.

[19] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 221–230, New York, NY, USA, 2011. ACM.

[20] Z. Xu. Directed test suite augmentation. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, 2011.

[21] Z. Xu et al. Directed test suite augmentaion: Techniques and tradeoffs. In *SIGSOFT FSE*, 2010.

[22] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, 2010.