High-Coverage Symbolic Patch Testing

Paul Dan Marinescu and Cristian Cadar

Department of Computing, Imperial College London
London, United Kingdom
{p.marinescu,c.cadar}@imperial.ac.uk

Abstract. Software patches are often poorly tested, with many of them containing faults that affect the correct operation of the software. In this paper, we propose an automatic technique based on symbolic execution, that aims to increase the quality of patches by providing developers with an automated mechanism for generating a set of comprehensive test cases covering all or most of the statements in a software patch.

Our preliminary evaluation of this technique has shown promising results on several real patches from the `lighttpd` web server.

1 Introduction

Writing correct software patches is a difficult task because developers have to ensure that new code not only executes correctly in itself but also interoperates flawlessly with already existing code, often written by other developers. That is, patches have to produce the expected behavioral changes without interfering with existing correct behavior, which is difficult to accomplish without thoroughly testing the patch. As a result, patches have a high rate of introducing failures [20,31,33]; for example, a recent study of software updates in commercial and open-source operating systems has shown that at least 14.8% to 25% of fixes are incorrect and have affected end-users [31].

Poor testing of patch code is therefore one of the main reasons for the high number of incorrect software updates. Most projects contain large amounts of untested code, and even those that come with regressions suites achieving relatively high patch coverage only test each statement on a limited number of paths and input values.

In this paper, we propose a practical technique that aims to improve the quality of software updates by providing an automatic way to generate test suites that achieve high coverage of software patches. Our approach is based on dynamic symbolic execution, a technique that has gathered a lot of attention recently, due to its ability to automatically generate complex test cases and find deep errors in the code under test.

In our approach, we enhance symbolic execution with the ability to focus on the lines of code affected by a patch. This is accomplished in two ways: first, we implement a new exploration strategy that prioritizes execution paths based on their distance from the patch, and has the ability to revert to an early state

A. Donaldson and D. Parker (Eds.): SPIN 2012, LNCS 7385, pp. 7–21, 2012.

when the patch cannot be reached under the current path condition. Second, we make use of existing concrete runs in order to quickly find program paths that reach, or are close to reaching, the patch. This latter enhancement is based on the observation that software developers sometimes accompany submitted patches with a test case that partially exercises the new code, and this test case could be easily used as a starting point for the symbolic exploration.

We envision a system in which program patches are comprehensively tested at the press of a button. For example, our approach could be embedded into a continuous integration system so that each update would trigger the generation of a set of test cases covering all or most of the statements in the patch being submitted.

The rest of the paper is structured as follows: Section 2 gives a high-level overview of our approach, while Section 3 presents our technique in more detail. Then, Section 4 discusses the most important implementation choices that we made, and Section 5 presents our preliminary results. Finally, we conclude with a discussion on related work in Section 6 and future work in Section 7.

2 Overview

The standard approach for ensuring that software updates do not introduce bugs is to enhance the program's regression suite with tests that cover the newly added code. The tests are either added in the same revision with, or shortly after, the patch; or, following agile development methods, before the patch is even written [2]. The technique is powerful because the tests encode developer knowledge about the program structure and past bugs, but requires discipline and manual effort to write the tests.

Symbolic execution-based testing, on the other hand, is an automatic technique that can systematically explore paths through a program, enabling high coverage test generation without manual effort. However, the technique inherently suffers from *path explosion*, the roughly exponential increase in number of execution paths through a program relative to the program size.

In this paper, we propose an approach that adapts symbolic execution to perform high-coverage regression testing. Our approach is motivated by the following two insights:

1. Testing patches requires exhaustive exploration of patch code but not of the entire program. The fact that patches are most of the time orders of magnitude smaller than the entire code base can be exploited to significantly improve testing scalability.
2. Developer regression tests usually execute the patch partially but do not cover all statements and corner cases. Therefore, we can effectively use them as a starting point for symbolic exploration, which then drives program execution towards uncovered code.

Our approach relies on prioritizing paths based on their distance from the patch code and on their compatibility with the patch context requirements, i.e. the conditions which have to hold in order for execution to flow towards the patch code. In addition, the approach leverages existing regressions tests to *seed* symbolic execution. The main contributions of our approach are:

1. A novel dynamic technique for guiding symbolic execution towards instructions of interest;
2. A prototype implementation of our technique based on the KLEE symbolic execution engine [6] and the ZESTI testing tool [19];
3. A case study of the application of our technique to real patches from the `lighttpd`[1] web server.

3 Design

The main goal of our technique is to construct program inputs that execute the uncovered parts of a patch. For the purpose of this presentation, we use line coverage as the coverage metric, but the technique can be easily adapted to other metrics as well. For simplicity of presentation, we consider that inputs are constructed for one instruction—*the target*—at a time. One can then simply apply the algorithm in turn to each uncovered instruction to cover the entire patch.

Synthesizing inputs which cover a target is an essential problem in automated test generation and debugging [1, 16, 27, 29, 30, 32]. While we borrow ideas from the state of the art in these areas, and combine symbolic execution, static analysis and various heuristics, our approach differs by treating the task as an optimization problem with the goal of exploring paths that minimize the estimated distance to the target. Given a suitable distance metric (which we discuss at the end of this section), a solution is found when minimizing the distance to zero.

Our patch discovery technique uses an iterative process, starting from an existing program input—*the seed*—obtained from the program test suite, standard symbolic execution, or a random input. For best results, this input should exercise instructions near the target. Intuitively, the approach attempts to steer execution off the path exercised by the seed input towards the target, guided by the estimated distance to it.

We begin by executing the program on the seed input, and remembering all branch points that depend on symbolic input, together with the symbolic path condition collected up to that point.[2] We refer to these branch points as *symbolic branch points*.

Then, at each iteration, we select the symbolic branch point whose unexplored side S is closest to the target (according to the estimated distance) and attempt to explore this side. If the current path condition allows S to be reached, we eagerly explore it, in what we call a *greedy exploration step*. Otherwise, if S is

[1] `http://www.lighttpd.net/`

[2] More details on how this process works can be found in [19].

```
1  void log(char input) {
2      int file = open("access.log", O_WRONLY|O_APPEND);
3      if (input >= ' ' && input <= '~') { // printable characters
4          write(file, &input, 1);
5      } else {
6          char escinput = escape(input);
7          write(file, &escinput, 1);
8      }
9      close(file);
10 }
```

Fig. 1. Example showcasing the greedy exploration step. Lines 5–8 represent the patch. Error handling code ommited for brevity.

```
1  if (0 == strcmp(requestVerb, "GET")) { ... }
2      . . .
3  for (char* p = requestVerb; *p; p++) {
4      log(*p);
```

Fig. 2. Example showcasing the execution regeneration step. As in Figure 1, the patch is on lines 5–8 of the `log` function.

infeasible under the current path condition, we enter the *informed path regeneration* mode, in which we travel back to the last symbolic branch point that made S unreachable and take there the other side of the branch. At this point, our current strategy is to explore the program path that preserves as much as possible from the initial path condition, in an attempt to quickly reach the desired branch side S. However, in future work, we plan to improve our technique by exploring multiple paths to S.

To illustrate our algorithm, we use the code snippet in Figure 1, which is based on a patch introduced in revision 2660 of `lighttpd`. The `log` function takes a single character as input and writes it into a text file. The function was initially always writing the character unmodified, but was patched in order to escape sensitive characters that could corrupt the log file. However, the program was tested only with printable character inputs and thus the patch was never executed. After seeding the analysis with such an input containing only printable characters, our technique determines that the `else` side of the symbolic branch point at line 3 is the unexplored branch side closest to the patch (in fact, it is part of the patch), and goes on to explore it (in a *greedy exploration step*) by negating the condition on line 3.

To understand when *informed path regeneration* is necessary, consider the example in Figure 2, in which the `log` function of Figure 1 is called for each character of the `requestVerb` string. Assuming that the seed request contains the `GET` verb, the comparison at line 1 constrains this input to the value `GET` for the remainder of the execution. Changing any of the characters in the `requestVerb` is impossible after this point because it would create an inconsistent execution,

and thus on this path we cannot follow the `else` side of the branch in the `log` function.

Instead, our *informed path regeneration* step travels back just before the execution of the symbolic branch point that introduced the constraint that makes the patch unreachable, and then explores the other side of that branch point. In our example, that symbolic branch point is the one at which `requestVerb[2]` was constrained to be `'T'`, and thus our technique takes here the other side of the branch, in which `requestVerb[2]` is constrained to be different from `'T'`. With this updated path condition, execution reaches again line 3 of the `log` function, where execution is allowed to take the `else` path and thus cover the patch.

We end this section with a discussion of our distance estimation function for the interested reader.

Our technique uses a context-sensitive, path-insensitive static analysis [21] to compute an approximation of the actual distance between two instructions. The distance is then further refined at runtime using callstack information. This analysis is used by KLEE itself to implement the search heuristic that minimizes the distance to an uncovered instruction, and works as follows. At the intra-procedural level, we define the distance between two instructions A and B contained in basic blocks BB_A, respectively BB_B as the minimum distance between BB_A and BB_B in the program control flow graph (CFG). Extending this definition to the inter-procedural level is not immediate; while edges can be introduced for function call instructions in the inter-procedural CFG, matching them statically with return edges is not trivial.

The solution to this problem involves two steps. First, we statically introduce two edges for each call instruction: one pointing to the called function with an associated weight of zero and another pointing to the instruction immediately following the call with a weight equal to the shortest path from the beginning to the end of the called function. These edges are modelling the two possible situations that can be encountered: the target is found before returning from the call or after. We call the resulting graph the *statically augmented CFG* and the shortest path between two of its nodes, their *static distance*.

Second, we add at runtime the return edges corresponding to the current call stack. While the resulting graph could be used directly to determine the minimum distance using a standard shortest path algorithm, this would add a significant overhead. Instead, we avoid running the full algorithm at runtime by observing that the target can be reached either by taking the shortest path in the statically augmented CFG or by returning from the current function and continuing on the shortest path. The minimum distance to the target is therefore the minimum between these two alternatives: the static distance to the target and the sum between the static distance to the closest return statement plus the distance from the associated call site to the target.

More formally, given the set of program instructions I, a callstack represented as an instruction vector $[I_1, I_2, \ldots, I_n] \in I^n$, the static distance from an instruction to the target $D : I \to \mathbb{N}$, and the static distance from an instruction to the

closest return instruction $R : I \to \mathbb{N}$, the context-sensitive minimum distance to the target is recursively defined as:

$$CSD([]) = \infty$$
$$CSD([I_1, I_2, \ldots, I_n]) = min(D(I_n), R(I_n) + CSD([I_1, \ldots, I_{n-1}])$$

Because functions D and R do not depend on the context, our analysis computes them once per program. The CSD is computed at each iteration of our algorithm for each candidate state, but note that the computation is independent of program size, depending linearly only on the size of the state's callstack.

4 Implementation

Our prototype implementation is built on top of the KLEE symbolic execution engine [6] and inherits the code responsible for combining concrete inputs with symbolic program exploration from ZESTI [19]. The `LLVM` infrastructure [17] is used to enable integration with KLEE and facilitate the static and dynamic analyses.

Compared to KLEE and ZESTI, our prototype implementation maintains only the last path explored instead of a tree containing all paths explored so far. While this simplifies the implementation, it makes our prototype miss targets which can only be reached via paths in which the distance to the target does not monotonically decrease, e.g., a target that is accessible only after a few iterations through a loop. We did not find such cases in the `lighttpd` revisions considered, but intend to handle this case in future work.

We also decided to execute at each iteration through our algorithm a batch of instructions, instead of a single one. This offers the advantage of generating more states from which to choose at the next iteration, with only a small time penalty, effectively providing a form of look-ahead. In certain scenarios, this compensates for the underestimation of the distance between two instructions, by permitting the execution of *longer* paths than dictated by the static estimation. Our implementation currently uses batches of 10,000 instructions during both the greedy exploration and the informed path regeneration steps.

5 Experimental Evaluation

We evaluated our protype implementation on the `lighttpd` web server, an efficient lightweight open-source server used by services such as YouTube and SourceForge. `lighttpd` is a mature system consisting at revision 2631—the earliest used in our experiments—of 37,517 effective lines of code, as reported by the `CLOC`[3] line counting tool, and containing a good test suite achieving 64.1% line coverage. We examined in detail three revisions from the last two years, period in which the number of lines of code and the coverage were largely unchanged. We ran all tests on a 64bit Ubuntu 10.04 i5-650 machine with 8GB of RAM.

[3] `http://cloc.sourceforge.net/`

Table 1. Patches examined in our evaluation; total effective lines of code (ELOC), ELOC covered by the regression suite, and ELOC covered by our tool. Revision 2660 contains 6 ELOC of dead code and 3 ELOC inaccessible in the test configuration.

Revision	ELOC	Covered ELOC	
		Regression test	Our tool
2631	20	15 (75%)	20 (100%)
2660	33	9 (27%)	24 (72%)
2747	10	4 (40%)	10 (100%)

In the following, we present three case studies in which we analyze the patches associated with `lighttpd` revisions 2631, 2660 and 2747. Our tool was able to cover all patch code accessible in the server test configuration. In the process, we found dead code in one of the patches, which turned out to be a bug. We reported the bug to the `lighttpd` developers, who promptly fixed it.[4]

The starting input for the analysis was manually chosen; we used the test case added with the patch for revision 2631 and a generic HTTP request from the `core-request.t` tests for the other two revisions, where no specific test existed.

Table 1 presents an overview of the three revisions, along with the number of new or modified effective lines of code (ELOC column). The lines are further placed into two categories: lines covered by the test suite and lines covered by our tool. It can be seen that for revision 2660, nine lines of code are not covered by our tool. Upon manual analysis, we discovered that six are dead code and three are unreachable in the server test configuration. Table 2 presents a summary of the additional code covered by our technique in each of the three revisions, with lines grouped by basic block. For each basic block, we report the number of iterations and the time needed to generate an input which covers it. We generate a total of 13 new inputs which added to the regression suite leave its execution time virtually unchanged at 6.6 seconds.

Revision 2631

Revision 2631 introduced the ability to handle requests for absolute URLs, e.g.:

```
GET http://www.example.com/ HTTP/1.0
```

The patch contains code which handles separately HTTP and HTTPS URLs but none of the existing regression tests contains absolute URLs. Furthermore, the test added with the patch only contains an HTTP request. Our tool successfully derives from it a new request for an HTTPS resource, exercising the previously uncovered code. We reproduce in Figure 3 the relevant part of the code.[5]

[4] See `http://redmine.lighttpd.net/issues/1551` for more details.

[5] The patch contains an additional, unrelated line of code not covered by the regression tests—line 566. This line was covered in our experiments by virtue of the modified order in which we sent requests to the server.

Table 2. Number of greedy and path regeneration iterations and time in seconds needed by our tool to generate inputs covering the lines of code not executed by `lighttpd`'s test suite. Lines are grouped by basic block.

Location (line numbers)	Greedy iterations	Path regeneration iterations	Time (seconds)
Revision 2631			
461	3	2	329
462,463,465	3	2	329
566	0	0	131
Revision 2660			
168	1	1	68
176,177	2	1	68
179,180	2	1	68
185,186	2	1	68
188,189	2	1	68
192-197	1	0	55
Revision 2747			
172	1	1	82
173	1	1	82
175,177	1	1	82
202,204	1	1	81

The target code is between lines 461 and 465—the other lines are already covered by the regression tests. We consider line 462 to show how our technique derives an input to cover new code from an existing test suite input. Table 3 presents the five derivation steps performed to transform the seed input `http://zzz.example.com/` into `https://zz.example.com/`, which covers the newly added code. We briefly explain how these steps relate to our algorithm.

(1) Our technique attempts to reach the `else` statement at line 460 and sets the 7th input character to an arbitrary value different from '/'. The string comparison on line 454 no longer returns 0 and the `else` branch is reached.
(2) Our technique attempts to satisfy the first part of the condition at line 460 and detects that the 5th input character must be 's'. However, it cannot directly set it to this value because it would create an inconsistent path; the `strncmp` function call at line 454 already compared this character to ':' and witnessed equality. Therefore, our technique travels just before this comparison and sets the 5th character to 's'.
(3) We continue to modify the input to satisfy the first part of the condition at line 460 and directly sets the 6th input character to ':'.
(4) We continue to modify the input to satisfy the first part of the condition at line 460 and directly sets the 7th input character to '/'.
(5) We continue to modify the input to satisfy the first part of the condition at line 460 and directly sets the 8th input character to '/'.

```
454  if (0 == strncmp(uri, "http://", 7) &&
455      NULL != (nuri = strchr(uri + 7, '/'))) {
456          reqline_host = uri + 7;
457          reqline_hostlen = nuri - reqline_host;
458
459          buffer_copy_string_len(con->request.uri, nuri, proto - nuri - 1);
460  } else if (0 == strncmp(uri, "https://", 8) &&
461      NULL != (nuri = strchr(uri + 8, '/'))) {
462          reqline_host = uri + 8;
463          reqline_hostlen = nuri - reqline_host;
464
465          buffer_copy_string_len(con->request.uri, nuri, proto - nuri - 1);
466  } else {
```

Fig. 3. Part of `lighttpd` revision 2631, which handles absolute request URLs. The patch is represented by lines 456, 457 and 460–465.

Table 3. Input derivation chain for covering the basic block containing target line 462 in `lighttpd` revision 2631. ↺ represents a path regeneration iteration and → represents a greedy iteration.

Step	Input	Type	Condition
	`http://zzz.example.com/`		
(1)	`http:/?zzz.example.com/`	↺	`url[6] != '/'`
(2)	`https/?zzz.example.com/`	↺	`url[4] == 's'`
(3)	`https:?zzz.example.com/`	→	`url[5] == ':'`
(4)	`https:/zzz.example.com/`	→	`url[6] == '/'`
(5)	`https://zz.example.com/`	→	`url[7] == '/'`

The input obtained after step 5 exercises the target line (462) and the algorithm terminates. As it can be seen, the newly found input is similar to the original, and not a pathological case, which we believe developers would prefer to incorporate into the regression suite.

Revision 2660

Revision 2660 was responsible for fixing a bug in the `accesslog` module. This module is responsible for logging all the requests made to the server so that they can be later viewed or processed by web analytics software. The log is maintained as text records, with space-separated fields. Remotely-provided data is quoted to allow automatic parsing of the file; this requires in turn special treatment of all quote (") characters. For example, a request with the referrer set to `foo" "bar"` would create the record:

```
127.0.0.1 - - [18/Apr/2012:02:14:44 +0100] "GET /index.html HTTP/1.0" 200
4348 "foo" "bar" "-"
```

```
165  if (str->ptr[i] >= ' ' && str->ptr[i] <= '~') {
166      /* printable chars */
167      buffer_append_string_len(dest, &str->ptr[i], 1);
168  } else switch (str->ptr[i]) {
169  case '"':
170      BUFFER_APPEND_STRING_CONST(dest, "\\\"");
171      break;
```

Fig. 4. Part of `lighttpd` revision 2660, which escapes senstive characters before logging them. The patch includes all of the lines shown.

The unpatched code detects a record with ten fields, the last three being `foo`, `bar` and `-`, while the correct interpretation is a record with nine fields, the last two being `foo" "bar` and `-`. The fix attempts to treat separately the quote and other control characters by escaping them. Figure 4 shows the relevant part of the patch. Printable characters are handled on line 167, on the `then` branch, while special characters are handled on the `else` branch. However, the `else` branch was not tested because no special characters were used in the regression tests. It turned out that the patch was incorrect because lines 170 and 171 are dead code; the quote character always satisfies the `if` condition on line 165 causing it to be always treated as a regular character. Another piece of dead code was handling the carriage return character, which cannot exist in the input because the request parsing code strips these caracters when breaking the request into lines.

Our tool covered all code accessible in the test server configuration, by generating appropriate HTTP requests. The rest of the code could have been reached by allowing our technique to change the server configuration file.

Revision 2747

Revision 2747 optimizes the `accesslog` module by introducing output buffering; instead of writing characters one by one, they are accumulated in an internal buffer and flushed when one of two events are encountered: a control character is logged or the end of the input is reached. Figure 5 shows the relevant code. As in the previous case, none of the regression tests contains control characters in the logged fields and the code associated with this event is never executed. Our tool successfully synthesizes the inputs needed to cover this code.

6 Related Work

Our technique fits within the paradigms of longitudinal and differential program analysis [22, 28], in which the testing effort is directed toward the parts of a program that have changed from one version to the next, i.e. software patches. In particular, differential symbolic execution [23] introduces a general framework

```
167  for (ptr = start = str->ptr, end = str->ptr + str->used - 1; ptr <
         end; ptr++) {
168      if (*ptr >= ' ' && *ptr <= '~') {
169          /* nothing to change, add later as one block */
170      } else {
171          /* copy previous part */
172          if (start < ptr) {
173              buffer_append_string_len(dest, start, ptr - start);
174          }
175          start = ptr + 1;
176
177          switch (*ptr) {
```

Fig. 5. Part of `lighttpd` revision 2747, which introduces output buffering for the access log. The patch includes all of the lines shown.

for using symbolic execution to compute the behavioral characterization of a program change, and discusses several applications, including regression test generation.

Xu and Rothermel [30] introduced directed test suite augmentation, in which existing test suites are combined with dynamic symbolic execution to execute uncovered branches in a patch. Given an uncovered branch $s_i \rightarrow d_i$ and a test case that reaches s_i, the technique uses dynamic symbolic execution to try to generate a test case that executes the branch, and then repeats this process until no more branches can be covered. The technique depends on the availability of tests that reach the source node of an uncovered branch and do not constrain the input to take only the already covered branch, while our approach tries to actively steer execution toward the patch by combining the greedy exploration and the informed path regeneration techniques.

eXpress [27] improves on directed test suite augmentation by pruning CFG branches which provably do not lead to the patch. While eXpress does not depend on having existing test cases that reach the source node of an uncovered branch and its algorithm allows it to prune significant parts of the search space, it does not actively try to steer execution toward the patched code. Statically-directed test generation [1] addresses this issue by guiding symbolic execution using a heuristic which includes the static instruction distance to the target and the size of the target's backward slice reachable from the current point. This heuristic roughly corresponds to our greedy exploration stage, but our approach is more robust due to its path regeneration component.

In addition to dynamic symbolic execution techniques to improve regression testing, the problem of generating inputs that reach a specific program point or execution path has been addressed through various other techniques and in different application scenarios. In the context of answering programmer queries regarding reachability, Ferguson and Korel [11] employ data dependence analysis to find test cases that reach a specified statement. They start with an initial

random guess, and then iteratively refine the guess to discover a path likely to hit the desired statement. Gupta et al. [14] use a combination of static analysis and generated test cases to hit a specified path. They define a loss function consisting of "predicate residuals" which roughly measures by "how much" the branch conditions for that path were not satisfied and then use a numerical solver to find test case values that can trigger the given path.

Research on automatic generation of filters based on vulnerability signatures [4, 5, 8, 9] addresses the problem of executing a specific target from a different angle. Given an existing input which exploits a program vulnerability, the goal is to infer the entire class of inputs which have the same behavior. Similarly, generating inputs with the same effect as a crashing input but which do not leak sensitive data, is used in bug reporting to preserve user privacy [7]. In the context of automated debugging, execution synthesis [32] and path optimization [16] attempt to solve a similar problem: generating an input or a path starting from a set of 'waypoints' through which execution has to pass.

While orthogonal to our approach, the software engineering community studied extensively test suite prioritization and selection techniques, e.g. [3,10,18,24]. These techniques are particularly useful for very large projects where running the entire test suite at each change of the system is infeasible (for example in the Windows operating system testing infrastructure [26]). Our approach is different in that it attempts to discover new test inputs but can leverage these techniques to choose the initial seeds.

Also orthogonal to our work, research on test suite augmentation requirements has used the differences between two program versions to derive requirements that test suites have to meet in order to ensure proper patch testing [15, 25]. While we currently only use simple coverage metrics to guide our analysis, it should be possible to combine our approach with such requirements.

A different approach for covering specific program points is to use genetic algorithms [12,29]. Such algorithms usually encode program paths as binary strings, each bit representing the outcome of a branch condition evaluation, and then define a fitness function and crossover and mutation operators operating on this encoding. While such algorithms proved effective when testing several protocols, it is unclear whether this approach yields good results on larger systems. The main concern is that the solution encoding does not naturally lend itself to effective crossover; in particular, given two paths which get close to the target (high fitness), alternating branch decisions from the first path with decisions from the second, does not generally yield a better path. Subsequent experiments [13] directly comparing genetic algorithms with directed search found that the latter generally performs better for this problem.

7 Discussion and Future Work

Motivated by the large number of buggy software patches, we have designed a new technique for patch testing, which successfully synthesized inputs to cover all accessible patch code in three case studies from the `lighttpd` web server. We

defined the code-covering challenge as an optimization problem for which we employed a novel heuristic based on two complementary components: a greedy path exploration, and an informed regeneration stage. The key aspect of our approach is the informed path regeneration stage, a technique that when the greedy exploration stage gets stuck, can derive a modified path which allows the greedy search to make progress.

Given the promising results obtained on our `lighttpd` case study, we plan to validate the algorithm by applying it to more patches across multiple systems. For best results, we also intend to remove all limitations discussed in the paper, in particular the ability to deal with the case when the static distance to the target must temporarily increase in order to reach the patch code.

Another aspect which we wish to address is automation. While our technique currently requires the manual specification of a seed input for each basic block in the patch, it would be desirable to automatically select the most promising input from the set of regression tests. This would be valuable for large patches, especially those touching multiple code areas. To this purpose, we envision leveraging test selection techniques [24]. Furthermore, we plan to improve our prototype by allowing it to automatically infer the patch location from a `diff` file.

Finally, we intend to evaluate the effectiveness of our technique in exercising uncovered program code. While we designed our approach for patch testing, one can immediately apply it to a standalone system version by considering all code not covered by the regression tests to be "patch".

Acknowledgments. We would like to thank the SPIN program committee chairs, Alastair Donaldson and David Parker, for their invitation to write this paper. We would further like to thank Alastair for his valuable comments on the text. This research has been supported by EPSRC through a DTA studentship and the grant EP/J00636X/1.

References

1. Babić, D., Martignoni, L., McCamant, S., Song, D.: Statically-directed dynamic automated test generation. In: Proc. of the International Symposium on Software Testing and Analysis, ISSTA 2011 (July 2011)
2. Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley (1999)
3. Binkley, D.: Semantics guided regression test cost reduction. IEEE Transactions on Software Engineering (TSE) 23(8) (1997)
4. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards automatic generation of vulnerability-based signatures. In: Proc. of the IEEE Symposium on Security and Privacy, IEEE S&P 2006 (May 2006)
5. Brumley, D., Wang, H., Jha, S., Song, D.: Creating vulnerability signatures using weakest preconditions. In: Proceedings of the 20th IEEE Computer Security Foundations Symposium, CSF 2007 (July 2007)

6. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008 (December 2008)
7. Castro, M., Costa, M., Martin, J.P.: Better bug reporting with better privacy. In: Proc. of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009 (March 2009)
8. Costa, M., Castro, M., Zhou, L., Zhang, L., Peinado, M.: Bouncer: securing software by blocking bad input. In: Proc. of the 21st ACM Symposium on Operating Systems Principles, SOSP 2007 (October 2007)
9. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: end-to-end containment of Internet worms. In: Proc. of the 20th ACM Symposium on Operating Systems Principles, SOSP 2005 (October 2005)
10. Elbaum, S., Kallakuri, P., Malishevsky, A.G., Rothermel, G., Kanduri, S.: Understanding the effects of changes on the cost-effectiveness of regression testing techniques. Software Testing Verification and Reliability 12 (2003)
11. Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Transactions on Software Engineering Methodology (TOSEM) 5(1), 63–86 (1996)
12. Godefroid, P., Khurshid, S.: Exploring very large state spaces using genetic algorithms. Int. J. Softw. Tools Technol. Transf. 6(2), 117–127 (2004)
13. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proc. of the Conference on Programing Language Design and Implementation, PLDI 2005 (June 2005)
14. Gupta, N., Mathur, A.P., Soffa, M.L.: Automated test data generation using an iterative relaxation method. In: Proc. of the ACM Symposium on the Foundations of Software Engineering, FSE 1998 (November 1998)
15. Gupta, R., Jean, M., Mary, H., Soffa, L.: Program slicing-based regression testing techniques. Software Testing Verification and Reliability 6, 83–112 (1996)
16. Lal, A., Lim, J., Polishchuk, M., Liblit, B.: Path Optimization in Programs and Its Application to Debugging. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 246–263. Springer, Heidelberg (2006)
17. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. of the International Symposium on Code Generation and Optimization, CGO 2004 (March 2004)
18. Li, Z., Harman, M., Hierons, R.M.: Search algorithms for regression test case prioritization. IEEE Transactions on Software Engineering (TSE) 33(4) (2007)
19. Marinescu, P.D., Cadar, C.: make test-zesti: A symbolic execution solution for improving regression testing. In: Proc. of the 34th International Conference on Software Engineering, ICSE 2012 (June 2012)
20. Mockus, A., Weiss, D.M.: Predicting risk of software changes. Bell Labs Technical Journal 5(2), 169–180 (2000)
21. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Publishing Company, Incorporated (2010)
22. Notkin, D.: Longitudinal program analysis. In: Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering, PASTE 2002 (November 2002)
23. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: Proc. of the ACM Symposium on the Foundations of Software Engineering, FSE 2008 (November 2008)

24. Rothermel, G., Harrold, M.J.: Analyzing regression test selection techniques. IEEE Transactions on Software Engineering (TSE) 22 (1996)
25. Santelices, R., Chittimalli, P.K., Apiwattanapong, T., Orso, A., Harrold, M.J.: Test-suite augmentation for evolving software. In: Proc. of the 23rd IEEE International Conference on Automated Software Engineering, ASE 2008 (September 2008)
26. Srivastava, A., Thiagarajan, J.: Effectively prioritizing tests in development environment. In: Proc. of the International Symposium on Software Testing and Analysis, ISSTA 2002 (July 2002)
27. Taneja, K., Xie, T., Tillmann, N., de Halleux, J.: eXpress: guided path exploration for efficient regression test generation. In: Proc. of the International Symposium on Software Testing and Analysis, ISSTA 2011 (July 2011)
28. Winstead, J., Evans, D.: Towards differential program analysis. In: Workshop on Dynamic Analysis, WODA 2003 (May 2003)
29. Xu, Z., Cohen, M.B., Rothermel, G.: Factors affecting the use of genetic algorithms in test suite augmentation. In: Proc. of the 12th Annual Conference on Genetic and Evolutionary Computation, GECCO 2010 (July 2010)
30. Xu, Z., Rothermel, G.: Directed test suite augmentation. In: Proc. of the 16th Asia-Pacific Software Engineering Conference, ASPEC 2009 (December 2009)
31. Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., Bairavasundaram, L.: How do fixes become bugs? In: Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering, ESEC/FSE 2011 (September 2011)
32. Zamfir, C., Candea, G.: Execution synthesis: A technique for automated software debugging. In: Proc. of the 5th European Conference on Computer Systems, EuroSys 2010 (April 2010)
33. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering (TSE) 28(2), 183–200 (2002)