KATCH: High-Coverage Testing of Software Patches

Paul Dan Marinescu
Department of Computing
Imperial College London, UK
p.marinescu@imperial.ac.uk

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

ABSTRACT

One of the distinguishing characteristics of software systems is that they evolve: new patches are committed to software repositories and new versions are released to users on a continuous basis. Unfortunately, many of these changes bring unexpected bugs that break the stability of the system or affect its security. In this paper, we address this problem using a technique for automatically testing code patches. Our technique combines symbolic execution with several novel heuristics based on static and dynamic program analysis which allow it to quickly reach the code of the patch.

We have implemented our approach in a tool called KATCH, which we have applied to all the patches written in a combined period of approximately six years for nineteen mature programs from the popular GNU diffutils, GNU binutils and GNU findutils utility suites, which are shipped with virtually all UNIX-based distributions. Our results show that KATCH can automatically synthesise inputs that significantly increase the patch coverage achieved by the existing manual test suites, and find bugs at the moment they are introduced.

Categories and Subject Descriptors

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

General Terms

Reliability, Verification
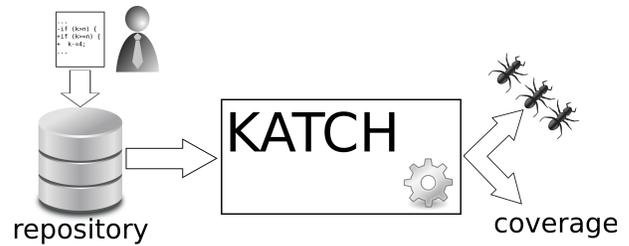
Keywords

Patch Testing, Symbolic Execution

Figure 1: KATCH is integrated in the software development life cycle and automatically generates inputs that execute newly added or modified code.

1. INTRODUCTION

A large fraction of the cost of maintaining software is associated with detecting and fixing errors introduced by recent patches. It is well-known that patches are prone to introduce failures [20, 42]. As a result, users often refuse to upgrade their software to the most recent version [11], relying instead on older versions which are frequently prone to critical bugs and have a reduced set of features.

In this paper, we aim to improve the quality of software patches by providing the means to automatically test them. Over the last years, we have seen significant advances in test generation techniques [3, 12, 17, 25, 31, 38, 43]; in particular, dynamic symbolic execution has proved to be a good fit for comprehensively testing real software [5–8, 14, 18, 19, 35, 37], through its ability to systematically explore different program paths, accurately reason about memory, and interact with uninstrumented code. The vast majority of work on dynamic symbolic execution has focused on "whole-program" testing, in which all parts of the program are treated equally. However, more recent work has looked at various forms of incremental or directed dynamic symbolic execution, where the testing effort is focused on code that has changed from one version to the next [2,27,32,33,36,41]. Despite this recent progress, we are still far away from the goal of quickly and automatically generating test cases that cover code changes in real programs.

This paper aims to improve the state of the art in the area by (1) developing novel techniques that can rapidly cover recently changed code, and (2) applying these techniques to a large number of indiscriminately-chosen patches (specifically to all patches written in a combined period of six years for `GNU diffutils`, `GNU binutils` and `GNU findutils`).

At a high level, we envision a system which we call KATCH,[1] that would be fully integrated in the software development cycle, as shown in Figure 1. When a new patch is sent to the repository, our system automatically explores paths through the patch code using dynamic symbolic execution augmented with several patch-aware heuristics, and provides to the developer a set of test inputs that achieve high coverage of the patch code (which could be added to the regression suite), and a report of any bugs introduced by the patch, accompanied by actual inputs that trigger them. To be adopted by developers, a system like KATCH has to meet several requirements: (1) it has to be easy-to-use, ideally fully automatic; (2) it has to be fast, to encourage developers to run it after every single commit; and (3) it has to demonstrate "value" by finding bugs and generating inputs that cover more code through the patch than existing manual test suites. In this paper, we provide some promising evidence that such a system could become a reality. In our experiments on all the patches written in a combined period of around six years for nineteen applications, KATCH was able to significantly increase the overall patch coverage and find fifteen distinct bugs, while spending only a relatively short amount of time per patch.

In summary, the main contributions of this paper are:

1. A technique for patch testing that combines symbolic execution with several novel heuristics based on program analysis that effectively exploit the program structure and existing program inputs;

2. A flexible system called KATCH, based on the state-of-the-art symbolic execution engine KLEE that implements these techniques for real C programs;

3. A thorough evaluation of our technique on all patches made to nineteen programs in the `GNU diffutils`, `GNU binutils` and `GNU findutils` application suites over a cumulative period of approximately six years.

The rest of this paper is structured as follows. We give a brief overview of KATCH (§2), describe our approach in detail (§3), and present the most important implementation details (§4). We then evaluate KATCH (§5), discuss related work (§6) and conclude (§7).

2. OVERVIEW

While the code of real software systems is frequently changing, these changes—or patches—are often poorly tested by developers. In fact, as we report in §5.1, developers often add or modify lines of code without adding a single test that executes them! To some extent, we have not found this result surprising, as we know from experience how difficult it can be to construct a test case that covers a particular line of code.

While the problem of generating inputs that cover specific parts of a program is generally undecidable, we believe that in many practical circumstances it is possible to automatically construct such inputs in a reasonable amount of time. Our system KATCH uses several insights to implement a robust solution. First, KATCH uses existing test cases from the program's regression suite—which come for free and
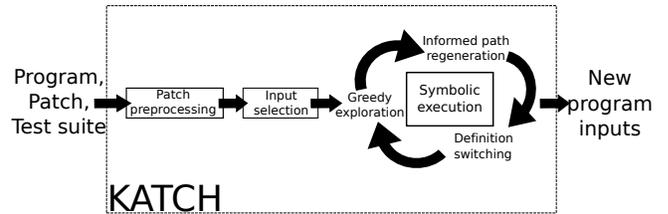

Figure 2: The main stages of the KATCH patch testing infrastructure.

often already execute interesting parts of the code—as a starting point for synthesising new inputs. For each test case input, KATCH computes an *estimated distance* to the patch and then selects the closest input (§3.2) as the starting point for symbolic exploration. Second, symbolic execution provides a framework for navigating intelligently through the intricate set of paths through a program, starting from the trace obtained by running the previously identified closest input. To reach the patch, KATCH employs three heuristics based on program analysis: greedy exploration, informed path regeneration (§3.3) and definition switching (§3.4).

Figure 2 presents the high-level architecture of KATCH. The framework takes as input a program, a set of existing program inputs and a patch description in the form of a `diff` file and automatically constructs new inputs that execute the patch code by following three steps.

Patch preprocessing is responsible for parsing the raw patch file and splitting it into lines of code. Lines of code that are part of the same basic block (and thus always executed together), are grouped to form a single *target*. Targets which are already executed by the program's regression suite are dismissed at this step. For each remaining target, the following stages are executed to synthesise an input which exercises it.

Input selection leverages the fact that real applications already come with regression suites that contain a rich set of well-formed inputs created by the developers. Input selection takes as input the program, a target and an existing test suite. It then associates with each of the test inputs a *distance* estimating the effort required to modify it such that it executes the target. The *closest* input to the target is then used in the next stage.

The last step combines *symbolic execution* with three heuristics based on program analysis to derive a new input that exercises the target, starting from the input selected at the previous step. The role of symbolic execution is twofold. First, it provides a framework for inspecting the program branch decisions and their relation to program inputs, and gives the means to generate new inputs by changing the outcome of particular branches. Second, it thoroughly checks program operations such as memory accesses and assertions, in order to find errors. The heuristics based on program analysis complement symbolic execution by partly mitigating its scalability problems and steering it actively towards the target.

To scale this process to multiple systems and hundreds or thousands of patches, we have also built an infrastructure which executes automatically, as appropriate, each of the previous steps, requiring no changes to the systems under test nor to their regression suites (§4).

---

[1] The name comes from K[LEE]+[P]ATCH. KLEE is an open-source symbolic execution engine on which KATCH is based.

3. KATCH

This section describes in more detail the KATCH patch testing infrastructure: patch preprocessing (§3.1), input selection with weakest preconditions (§3.2), greedy exploration with informed path regeneration (§3.3) and definition switching (§3.4).

3.1 Patch Preprocessing

The first stage of our analysis is mainly responsible for retrieving each program version from the version control system, determining the differences from the previous version—i.e. the patch—and breaking this patch into lines which are then handled individually by the subsequent steps. In addition, the lines are filtered and consolidated when appropriate, as described next.

While each line in a patch is a potential target to KATCH, in practice, many lines do not need to be considered. First, source code contains many non-executable lines, such as declarations, comments, blank lines, or lines not compiled into the executable due to conditional compilation directives. Second, lines which are part of the same basic block are always going to be executed together, so we only need to keep one representative per basic block. Finally, we are not interested in lines already covered by the system's regression test suite.

The patch preprocessing stage is responsible for eliminating all these lines and works in two steps: a first step performs a simple static pass to eliminate non-executable code and all but one line in each basic block, and a second step runs the program's regression suite to eliminate lines already covered by its test cases.

This results in a set of lines which are on the one hand executable and on the other hand are not executed by the program's test suite—which we call *targets*. Each of them is processed individually in the following stages.

3.2 Seed Selection with Weakest Preconditions

Our input synthesis technique starts from an existing program input—called *the seed*—extracted from the program's test suite, and iteratively changes it. The ideal seed executes code which is *close* to the target, in order to allow KATCH to quickly steer execution by switching only a few branch outcomes to reach the target.

To estimate the distance between the path executed by a seed and the target, we calculate the (static) minimum distance in the program's interprocedural control flow graph (CFG) between each basic block exercised by the seed and the target basic block. Intuitively, the effort of symbolic execution lies in switching the outcome of branch statements, therefore we compute this distance in terms of the number of branch statements between the two basic blocks.

We also simplify the minimum distance computation by not requiring it to be context-sensitive. To do this, we note that pairs of matched function calls and returns should not contribute to the distance between two basic blocks. In practice, this means that we can "shortcut" function calls, i.e. each function call introduces an edge to the instruction immediately following the call, in addition to the edge to the target function. In turn, shortcutting function calls allows us to remove all return edges, simplifying the analysis.

However, the estimated distance outlined so far—which we call *C-flow* distance, as it only takes the control flow into account—can select inputs which exercise paths close

		C−flow	WP
1	**if** (input < 100)	2	4
2	f(0);	1	4
3			
4	**if** (input > 100)	3	3
5	**if** (input > 200)	2	2
6	f(input)	1	1
7			
8	**void** f(**int** x) {		
9	**if** (x == 999)	1	1
10	*// target*	0	0
11	}		

Figure 3: Code snippet with instructions annotated with the minimum distance to the target, computed using only control-flow analysis (C-flow column) and control-flow analysis combined with our weakest precondition variant (WP column).

to a target, but cannot be easily changed to actually reach the target. In the interest of simplicity, we show a contrived example in Figure 3 to illustrate such a scenario. The code snippet takes a single integer as input and uses it to decide whether to call function f, which contains the target. The only input which exercises the target is 999. The figure also shows the C-flow distance from each instruction to the target. For example, the C-flow distance for the instruction at line 5 is 2, because the shortest path to the target traverses two branches (on lines 5 and 9).

For simplicity, assume that we only want to assess whether input 50 is better than input 150. From a pure control-flow perspective, 50 appears to be a better choice because it exercises function f and gets to the if condition guarding the target (while 150 does not call f at all). Upon closer inspection however, it is clear that the target guard condition x == 999 is always false on this path because function f is called with argument 0 on line 2, and therefore the target cannot get executed through this call. This observation led us to create a technique which automatically prunes CFG edges which can be proven to make the target unreachable.

To find such edges we use an interprocedural data-flow analysis which computes for each target and basic block in the program a necessary condition to reach that target, a form of weakest preconditions [13]. If by traversing an edge we obtain a false condition, we conclude that the target is unreachable through that edge. Considering the same example, the branch on line 9, which guards the target, creates the condition x = 999, while the edge from the function call at line 2 defines x to be 0. By substitution, we obtain the formula 0 = 999 which evaluates to false, and conclude that the function call on line 2 cannot help in reaching the target. Column *WP* of Figure 3 shows the minimum distance from each instruction to the target after removing the edge introduced by this function call. Lines 1 and 2 have their distances updated.

For the interested reader, we present the data-flow equations which compute the preconditions, relative to a target, at the beginning and at the end of each basic block, and give an intuition on their correctness.

(1) $\quad out_b = \bigvee_{s \in succ_b}(cond_{b \to s} \wedge in_s)$

(2) $\quad in_b = wp(b, out_b)$

With initial values:
$$in_{target} = true, out_{target} = false$$
$$in_b = out_b = false, \forall b \neq target$$

$cond_{b \to s}$ represents the condition required to go from basic block b to s. For unconditional branches the condition is always `true`. $wp(b, out_b)$ is the standard weakest precondition function, applied to basic block b and postcondition out_b, which is easily computed for a single basic block as we describe below.

The equations guarantee that any edge $b \to s$ for which $cond_{b \to s} \wedge in_s$ is `false` and any basic block b for which in_b is `false` cannot lead to the target.

The first equation intuitively says that at the end of a basic block b, the condition to reach the target is the disjunction of the conditions for all possible paths from that basic block to the target. The second equation obtains the weakest precondition for a basic block from its corresponding postcondition. This is done by iterating through the instructions of the basic block in reverse order and substituting all variables from the postcondition with their definition, as appropriate. A variable not defined in the current basic block is left unchanged. When applied to the target basic block, the wp function always yields `true`.

Solving the system is done using a standard fixed-point computation approach. Our implementation makes two conservative approximations to make the analysis tractable even on large programs. First, the wp function only handles assignments. If the basic block applies other operations to the postcondition variables, the returned value is `true`. Second, a disjunction of syntactically non-identical formulae in the first equation is also treated as `true`, to prevent formulae from growing exponentially.

These two approximations capture two common practical cases. First, formulae which become `false` when applying the wp function usually correspond to code patterns which use boolean flags or enumerated type variables in branch conditions; basic blocks which, for example, set a flag to `false` and make a certain branch infeasible are recognised accordingly. The example in Figure 3 is such a case.

Second, formulae may become `false` because the set of conjuncts accumulated through the first data-flow equation becomes inconsistent. This case correspond to patterns where the same variables are used in branch conditions multiple times, possibly in different parts of the program and some of the conditions are mutually incompatible. A simple example can be observed in Figure 3: the weakest preconditions algorithm can prove that the branch between lines $1 \to 2$ does not to lead to the target because the branch condition $input < 100$ is incompatible with the condition $input > 100$ which appears subsequently on the only path to the target.

After obtaining the distance from each basic block to the target through the control-flow analysis, and the branches and basic blocks that cannot lead to the target through the weakest preconditions analysis, the distance from each available seed input to the target is computed as follows:

1. Compute the subgraph G of the program's CFG induced by running the program on the seed input. G's nodes are the basic blocks executed and its edges are the branches taken during execution;

2. Remove all nodes and edges in G which were proven to make the target unreachable;

3. Iteratively remove from G all nodes *orphaned* by the previous step, i.e. while there are nodes with in-degree 0 (except the program entry point), remove them and all their outgoing edges;

4. Choose the minimum from the distances of the remaining nodes to the target.

3.3 Greedy Exploration with Informed Path Regeneration

The last and most challenging stage of KATCH is responsible for transforming the previously selected seed input into a new input that executes the target. Our approach is based on symbolic execution [24], a program analysis technique that can systematically explore paths through a program. The key idea behind symbolic execution is to run the program on *symbolic input*, which is initially allowed to have any value. Then, whenever a branch depending directly or indirectly on the symbolic input is encountered, execution is conceptually forked to follow both sides if both are feasible, adding appropriate constraints on each side of the branch. Finally, whenever a path terminates or hits an error, the constraints gathered on that path are solved to produce a concrete input that exercises the path. For example, if we run the code in Figure 3 treating the `input` variable as symbolic, then at branch 1 execution will be split into two paths: one following the `then` side of the branch, on which we add the constraint that `input < 100`, and one following the implicit `else` side of the branch, on which we add the constraint that `input ≥ 100`. When the path with the constraint `input < 100` reaches line 4, only the `else` side is feasible, so no other path is spawned at this point. On the other hand, when the path with the constraint `input ≥ 100` reaches line 4 both sides are feasible, so execution is again split into two paths, one on which we add the constraint that `input > 100`, and one on which we add the constraint that `input ≤ 100` (which together with the existing constraint that `input ≥ 100` gets simplified to `input = 100`). The branches at lines 5 and 9 similarly spawn new execution paths. Finally, when a path terminates, a constraint solver is used to generate a solution to all the constraints gathered on that path, which represents an input that can be used to exercise the path. For example, the path with the constraints `input ≥ 100`, `input > 100` and `input ≤ 200` may return the solution `input = 150` which exercises that path.

In our approach, we start symbolic execution from an existing input, the seed, similarly to the approach taken in concolic execution [18, 35] and our ZESTI system [29]. The seed is then iteratively modified by exploring paths which get closer to the target; symbolic execution provides the framework for the exploration and constraint solving is used to map program paths back to inputs. The novelty of our approach lies in the way paths are selected for exploration.

The selection is based on a metric which estimates the distance from a path to the target, similar to the distance used by the input selection stage (§3.2). In each iteration, we execute the program using the latest input, and remember all branch points, e.g. `if` conditions, along with information necessary to continue execution on the other side of the branch, should we later decide to.

We then select the branch point whose unexplored side S is closest to the target (according to the estimated distance) and attempt to explore this side. If S is feasible,

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

Figure 4: Example based on `lighttpd` patch `2660` used to illustrate the greedy exploration step. Lines 3, 5–8 represent the patch.

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

Figure 5: Example based on `lighttpd` patch `2660` used to illustrate the informed path regeneration step. As in Figure 4, the patch is on lines 3, 5–8 of the `log` function.

i.e. the conjunction of the branch condition towards S and the current path condition is satisfiable, we eagerly explore it, in what we call a *greedy exploration step*. Otherwise, we examine two possibilities: (1) the branch condition is symbolic, i.e. it has a data dependence on program input on the current path and (2) the branch condition is concrete, i.e. it has a control dependence on program input. Informally, a branch condition is data dependent on program input if data propagates from the input to at least one of the variables involved in the branch condition via a sequence of assignments. A condition is control dependent on the input if at least one variable involved in the condition has more than one reaching definition. Note that some conditions can be both data and control dependent.

For data dependent conditions (including those which are also control dependent), we apply *informed path regeneration*, where we travel back to the branch point that made S infeasible and take there the other side of the branch. For control dependent conditions, we attempt to find a different definition for the variables involved in the condition, such that the condition becomes `true`. In the following, we examine each of these cases in detail.

To illustrate our approach, we use the code snippet in Figure 4, which is based on a patch introduced in revision `2660` of the `lighttpd` web server, which we analysed in prior work [28]. The `log` function takes a single character as input and writes it into a text file. The function was initially writing all characters unmodified, but was patched in order to escape sensitive characters that could corrupt the file structure. However, the program was tested only with printable character inputs and thus the `else` branch 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 5, in which the `log` function of Figure 4 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.

3.4   Definition Switching

Informed path regeneration does not work if the branch condition has a concrete value, essentially because we cannot reason symbolically about concrete expressions. This case occurs when the condition does not have a data dependence on the input on the currently explored path, but only a control dependence. Figure 6, containing code from `diffutils` revision `8739d45f`, showcases such a scenario. The revision modifies line 235, which is our target.

To execute the patch, one needs to pass through the `switch` statement on line 230, requiring `ig_white_space`, and in turn `ignore_white_space` to be equal to the `IGNORE_ALL_SPACE` constant. This only happens when the program is given the `-w` command line argument (line 495). Assuming the current input does not include `-w`, the lack of a data dependence between the `switch` condition and the command line arguments renders informed path regeneration unusable. To solve this problem, we use a lightweight approach that finds the reaching definitions for the variables involved in the condition using static analysis and then attempts to find a path to an uncovered definition using the two techniques previously presented. To further improve the chances of getting the right definition early, the algorithm gives priority to definitions that can be statically shown to satisfy the target branch condition. Furthermore, the algorithm works recursively on all definitions which were already executed, but for which the right-hand side is not a constant. That is, the algorithm can be nested multiple times by using a stack of intermediary targets; when a definition needs to be switched, the active target is saved on the stack and the selected definition becomes the new active target. As soon as the definition is executed, the previous target is popped off the stack.

To show how definition switching works in practice, consider the same code snippet and the input `-a -y - a b` provided by input selection, which compares two files `a` and `b` treating them as text (`-a`), and outputs the results side-by-

src/io.c

```
217   enum DIFF_wh_sp ig_white_space = ignore_white_space;

...

230   switch (ig_white_space)
231     {
232     case IGNORE_ALL_SPACE:
233       while ((c = *p++) != '\n')
234         if (! isspace (c))
235           h = HASH (h, ig_case ? tolower (c) : c);
236       break;
```

src/diff.c

```
291   while ((c = getopt_long (argc, argv,
                    shortopts, longopts, NULL)) != −1)
292     {
293     switch (c)
294       {
...
319       case 'b':
320         if (ignore_white_space < IGNORE_SPACE_CHANGE)
321           ignore_white_space = IGNORE_SPACE_CHANGE;
322         break;
323
324       case 'Z':
325         if (ignore_white_space < IGNORE_SPACE_CHANGE)
326           ignore_white_space |= IGNORE_TRAILING_SPACE;
...
389       case 'E':
390         if (ignore_white_space < IGNORE_SPACE_CHANGE)
391           ignore_white_space |= IGNORE_TAB_EXPANSION;
392         break;
...
494       case 'w':
495         ignore_white_space = IGNORE_ALL_SPACE;
496         break;
```

Figure 6: Example from `diffutils` revision `8739d45f` showcasing the need for definition switching. The patch is on line 235 and is guarded by a condition that is control dependent on the input.

side (`-y`). This input reaches the guarding switch statement on line 230 but evaluates to a different case. To reach the target, we need to modify the input such that the condition `ig_white_space == IGNORE_ALL_SPACE` is satisfied. Because the condition does not have a data dependence on the input, KATCH attempts to find another definition for the `ig_white_space` local variable and discovers one on line 217. However, it detects that this definition was already executed, so it recursively attempts to find definitions for the right-hand side of the assignment, the `ignore_white_space` global variable.

At this point, KATCH finds four definitions, each corresponding to a different command line argument and decides to use `ignore_white_space = IGNORE_ALL_SPACE` because it matches exactly the original condition which it attempts to satisfy. KATCH now pushes the original target (line 235) to the stack and changes the active target to line 495. It then

```
REPO="git://git.savannah.gnu.org/diffutils.git"
DIFFTARGETS="src lib"
PROGRAMS="src/diff src/diff3 src/sdiff src/cmp"
LIBS="-lrt"
```

Figure 7: Configuration file used to test `diffutils`. The file specifies the repository address, the folders which may contain relevant changes, the programs to test and the libraries required to build the system.

uses an informed path regeneration step to replace the first command line argument with the required `-w` option. This reaches the intermediary target which causes the original target to be popped off the target stack and transformed back into the active target. Execution continues and this time the `ignore_white_space` and `ig_white_space` variables have the appropriate values to reach the patch. The synthesised input which reaches the patch is `-w -y - a b`.

4. IMPLEMENTATION

KATCH consists of patch preprocessing scripts, the input selection subsystem, the augmented symbolic execution tool and a set of scripts which automatically iterate through all patches in a given set of program revisions. Most components operate at the level of LLVM bitcode, the intermediate language used by the popular LLVM compiler [26].

At a high level, a tester is only required to create a configuration file with details about the system to test, such as the repository address and the names of targeted executable files. Figure 7 shows the actual file used for testing `diffutils`. Optionally, the tester can also provide scripts for compiling the system and running its regression suite. Otherwise, the default `configure`, `make` and `make check` commands are used, adapted for creating LLVM bitcode along with the native executables. Having this setup, the tester only needs to issue a command such as:

```
./test-patch-multiple diffutils rev1 rev2
```

to test all `diffutils` revisions between *rev1* and *rev2*. This script could be easily added to a continuous integration system to automatically test the last patch.

4.1 Patch Preprocessing

Patch preprocessing is implemented via two LLVM passes: the first one statically prunes non-executable lines by traversing the compiled program and using debug information to map LLVM instructions back to source code; a line is deemed non-executable if no instruction maps back to it. The second pass instruments the program to obtain test suite coverage information and determine which patch lines are executed by the test suite.

4.2 Input Selection

Input selection uses a combination of scripts and LLVM passes to instrument the program and analyse the execution of its test suite. In this phase, the original executables specified in the configuration file are replaced with wrapper scripts that invoke an instrumented copy of the corresponding binary. For each target, the instrumentation computes and outputs to a file the minimum distance from each test suite input, allowing the wrapper to determine which input

gets closest to the target. This input is identified transparently by its *sequence number*, i.e. the number of times the program was executed by the test suite so far. Subsequently, we run the test suite again and when reaching the target sequence number, we invoke KATCH instead of the regular executable.

The only assumption made by our approach is that the order of running the tests is deterministic, which holds in all cases we have looked at. While we could have used other solutions, we found that they are either not as general or they do not perform as well. For example, a different solution would be to record the program arguments used to get to the minimum distance instead of the sequence number and then run KATCH directly using these arguments. However, this approach fails when the test suite harness creates any non-trivial setup, not captured by the command line arguments, such as files, pipes or environment variables. Another approach is to directly run KATCH on all test inputs. The downside is the larger overhead: symbolically interpreting the program is several orders of magnitude slower than native execution, while the instrumented programs have a comparable execution time to their native counterparts.

Instrumenting the program is performed through an LLVM pass which takes as input the original program and the current target. The pass uses a standard shortest path algorithm to statically compute the distance from each basic block to the target in the program's interprocedural control flow graph and adds code to each basic block to record this distance at runtime. It further uses the weakest precondition data-flow analysis described in §3.2 to refine this distance and inserts code in the executable to eliminate from the computation those branches which provably cannot lead to the target. To increase maintainability, most of the instrumentation is written in C++ as a set of helper functions which are then statically linked with the target program.

4.3 Symbolic Exploration

KATCH is implemented on top of the KLEE [6] open-source symbolic execution engine. KATCH starts by executing the program on the path induced by the selected seed input to completion or until a predefined timeout expires. On this path it records all possible branches, feasible and infeasible, that the program does not take. This provides more information for selecting the next path, as opposed to previous approaches which only considered the feasible branches. The branches are then considered in order of increasing distance to the target as candidates for one of the techniques employed by KATCH: greedy exploration for feasible branches, and informed path regeneration or definition switching for infeasible branches. Once a suitable branch is found, the process repeats, executing a batch of instructions and re-evaluating the available paths.

We decided to use a batch of instructions, instead of a single one because this offers the advantage of generating more paths to choose from 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 LLVM instructions.

KATCH uses another optimisation to handle efficiently several common functions whose use is expensive in a symbolic execution context: the `getopt` family of functions, and `strcmp`. For space reasons, we discuss only `getopt` below. The `getopt` functions are helpers used by many programs to process command line arguments. They work by allowing the programmer to write a simple specification of the arguments accepted by the program, thus moving the bulk of the command line parsing code inside the library functions. KATCH is aware of the `getopt` semantics and uses this information to speed up processing. More precisely, whenever the return value of `getopt` is a reaching definition, instead of recursively descending in the function code, it inspects the function argument corresponding to the specification of accepted command line arguments and directly determines the command line option needed to obtain the desired definition. The new argument is added to the command line and program execution restarts from the beginning.

4.4 Limitations

We discuss below the most significant limitations of our current prototype. Most importantly, we currently do not handle targets which are accessible only through function pointer calls that have not been exercised by the regression suite. Such indirect calls pose problems both during the static analysis when computing the closest input, and during dynamic exploration. The problems could be mitigated by including support for pointer analysis [1, Chapter 12] which KATCH currently does not offer.

Second, our current implementation of definition switching does not support aggregate data types such as structures and arrays. Finally, KLEE's environment model is incomplete, e.g. it does not handle certain system calls.

5. EXPERIMENTAL EVALUATION

For an objective evaluation of our technique, we have set ourselves the following two requirements. First, we have decided to do no cherry picking: once we have chosen a set of benchmark programs, rather than selecting the 10 (or 20, or 30) patches on which our technique works well, we included all the patches written over an extended period of time. Second, we have decided to allow a short timeout for our system, of no more than 15 minutes, which we believe is representative for the amount of time developers typically dedicate to testing a patch.

We evaluated KATCH on nineteen programs from the GNU `diffutils`, GNU `binutils` and GNU `findutils` systems. These are all mature and widely used programs, installed on virtually all UNIX-based distributions.

GNU findutils is a collection of three programs, `find`, `xargs` and `locate`. They are smaller in size than the other two benchmarks, having a combined 14,939 lines of code (LOC)[2] in the tools themselves, and include additional portions of code from `gnulib`, which totals more than 280,000 LOC at the latest revision that we inspected. We examined the 125 patches written in the two years and two months period between November 2010 and January 2013.

GNU diffutils comprises four programs, `diff`, `sdiff`, `diff3` and `cmp`. They are of medium size, with 42,930 LOC in the tools themselves, and include additional portions of code from `gnulib`, similarly to `findutils`. We have analysed all the 175 patches written during the 2.5 years between November 2009 and May 2012.

---

[2]We report the number of LOC in the latest version tested, measured using `cloc` (http://cloc.sourceforge.net).
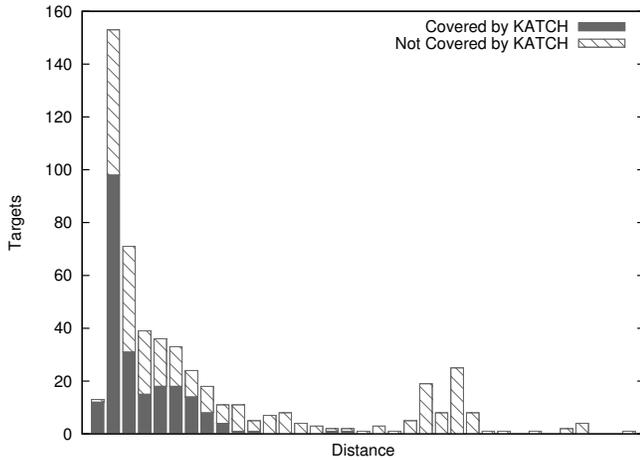
Figure 8: Distribution of minimum distances for the 939 targets not covered by the regression suites. The figure does not include 389 `binutils` targets accessible only through indirect function calls not exercised by the test suite, which are outside the current capabilities of KATCH. Each bar also shows the fraction of target covered by KATCH.

GNU binutils includes a variety of programs out of which we selected the twelve assorted binary utilities from the `binutils` folder (`addr2line`, `ar`, `cxxfilt`, `elfedit`, `nm`, `objcopy`, `objdump`, `ranlib`, `readelf`, `size`, `strings` and `strip`). They contain 68,830 LOC, and use the statically linked libraries `libbfd`, `libopcodes` and `libiberty`, which total over 800,000 LOC. Because of the more accelerated development pace in `binutils`, we examined a shorter 1 year 4 months period between April 2011 and August 2012, in which 181 patches were added to the `binutils` directory.

We set a short timeout of ten minutes per target for `findutils` and `diffutils` and a timeout of fifteen minutes for the larger `binutils` programs. We used a four-core Intel Xeon E3-1280 machine with 16 GB of RAM, running a 64-bit Fedora 16 system. As an extra safety check, we verified that all inputs generated by KATCH execute the corresponding patch code on the natively compiled programs, using `gcov` for coverage measurement.

Our tool and results have been successfully evaluated by the ESEC/FSE artifact evaluation committee and found to meet expectations.

5.1 Coverage Improvement

As a first measure of KATCH's effectiveness, we looked at its ability to improve *patch coverage*. Because KATCH operates at the basic block level, we define patch coverage as the number of executed basic blocks which contain statements added or modified by a patch over the total number of basic blocks which contain such statements.

The patches analysed contain altogether 9,873 textual lines of code.[3] After processing these lines to remove non-executable statements and group related executable lines, we obtained 1,362 potential targets which are part of 122 patches. Upon manual inspection, we found that the rest of the patches only keep the build system up-to-date with the program dependencies, or make changes to

---

[3]This includes only lines in `.c` and `.h` files.

Table 1: Number of targets covered by the manual test suite, and the manual test suite plus KATCH.

Program Suite	Targets	Covered	
		Test	Test + KATCH
`findutils`	344	215 (63%)	300 (87%)
`diffutils`	166	58 (35%)	121 (73%)
`binutils`	852	150 (18%)	285 (33%)
Total	**1,362**	**423 (31%)**	**706 (52%)**

the documentation or test suite. A total of 423 targets were already covered by the system's test suite, leaving 939 targets for KATCH to analyse.

The first step performed by KATCH is to compute the minimum distance from the regression test inputs to each target. Figure 8 presents the distribution of the minimum distances, which also provides a rough estimate of the work that KATCH needs to do for each target. More than half of the targets have regression tests which get relatively close to the target, at a distance smaller than five. Just a small fraction of the targets are at a distance over 20, which are all contained in completely untested `binutils` features. The figure does not include 389 `binutils` targets accessible only through indirect function calls not exercised by the test suite, which are outside the current capabilities of KATCH.

Table 1 summarises the results obtained after applying KATCH to these 939 targets. The *Targets* column lists the total number of targets for each benchmark and the *Covered* column lists the number of targets covered by the regression test suite, respectively the regression test suite and KATCH. It can be seen that KATCH has automatically increased the overall patch coverage from 31% to 52% (covering 283 out of the 939 targets).

We analyse below the cases in which KATCH fails to reach the target, in order to illustrate its limitations. More than half of the cases are targets accessible only through indirect function calls never exercised by the test suite, which our current prototype does not handle (see §4.4).

Another large number of cases relate to complex or multiple guard conditions. To satisfy them, KATCH would need to alter the input structure or to have access to a richer test suite, containing different seed inputs. For example, many `binutils` targets are only executed when the input file contains specific sections, with an individually defined structure. When none of the test suite files contains such a section type, the targets are usually not covered because KATCH cannot synthesise a complete section from scratch in the allotted time.

A more subtle scenario involves data stored using variable-length encoding, which is often used by `binutils`. In this case, KATCH can easily change input values only as long as they would be encoded using the same length. Changing to a value with a different encoding length would require inserting or removing one or more bytes in the middle of the input, significantly increasing complexity by possibly affecting other parts of the input such as header offsets.

Therefore, KATCH works best when the seed input does not need to have its structure altered. This is an inherent limitation of symbolic execution, which does not treat the input structure (e.g. its size) symbolically. This limitation is mitigated as the test suite quality improves and the chances of finding a good seed input increase.

Table 2: Number of targets covered by different combinations of heuristics: greedy (G), greedy and informed path regeneration (G+IPR), greedy and definition switching (G+DS) and all (KATCH).

Program Suite	G	G + IPR	G + DS	KATCH
`findutils`	74	85	78	85
`diffutils`	25	29	49	63
`binutils`	70	121	76	135
Total	**169**	**235**	**203**	**283**

The fact that our definition switching analysis does not support aggregate data types (§4.4) also affects several targets. A smaller number of targets cannot be reached due to the incomplete environment model implemented in KLEE, such as unsupported system calls.

Finally, we also noticed that several targets were not covered because they correspond to unreachable code on our test system—e.g. are reachable only on operating systems which differentiate between text and binary files.

In addition to the overall coverage improvement, we also wanted to measure exactly the contribution of each heuristic used by KATCH. We therefore re-executed the same experiments, selectively disabling all possible combinations of heuristics (note that all heuristics depend on greedy). Table 2 shows the results. It can be seen that the improvement brought by each heuristic varies from system to system. At one end of the spectrum `diffutils` covers 152% more targets when using all heuristics compared to greedy alone, while at the other end `findutils` sees only a 15% improvement. Overall, informed path regeneration and definition switching combined brought a 67% improvement.

We have also run our experiments using KLEE instead of KATCH, to see how well a pure dynamic symbolic execution approach performs. We ran KLEE for 30 minutes on each revision, and we used an appropriate set of symbolic arguments. The results were very poor, with only two targets covered in the smaller `findutils` programs.

5.2 Bugs Found

KATCH was also able to identify a total of fifteen distinct crash bugs. We could verify that thirteen of these are also present in the latest version and we reported them to the developers, providing automatically-generated inputs which trigger them. Eleven of the bugs were discovered as a direct consequence of KATCH's goal to reach the target: six bugs are in the actual targets and are discovered as they are introduced, while the other five are discovered because code is introduced in their vicinity.

One bug was found in `findutils`, and the rest were found in `binutils`, the largest and most complex of all three application suites. A manual analysis of the bugs revealed that they relate to the handling of unexpected inputs. Interestingly, `binutils` generally does a good job handling such situations, but in several cases, the checks performed are incomplete. An example is bug 15206[4] in `objdump`, a buffer overflow caused by improperly checked buffer bounds. The bug appears in revision `119e7b90`, shown in part in Figure 9.

---
[4] http://sourceware.org/bugzilla/show_bug.cgi?id=15206

binutils/dwarf.c

```
243    process_ext_line_op (unsigned char *data, int is_stmt)
...
251        len = read_leb128 (data, & bytes_read, 0);
252        data += bytes_read;
...
380        unsigned int rlen = len − bytes_read − 1;
...
391        for (; rlen; rlen−−)
392            printf (" %02x", *data++);
```

Figure 9: Example showing a bug found by KATCH, introduced in `binutils` revision `119e7b90`. The bug is triggered on line 392. The highlighted lines are part of the patch.

binutils/readelf.c

```
12232    while (external < (Elf_External_Note *) ((char *)
             pnotes + length))
12233    {
...
12238        if (!is_ia64_vms ())
12239        {
12240            inote.type = BYTE_GET (external−>type);
12241            inote.namesz = BYTE_GET (external−>namesz);
```

Figure 10: Example showing a bug found by KATCH, introduced in `binutils` revision `b895e9d`. The bug is triggered on line 12240. The highlighted line is part of the patch.

Line 251 reads the buffer size from the buffer itself and lines 391 and 392 rely on this size to iterate through the entire buffer. The overflow occurs if the size read does not match the allocated buffer size.

Another example is the `readelf` bug 15191,[5] shown in Figure 10. This bug was detected in revision `b895e9d`, when code was added to conditionally execute several existing lines. None of the code shown was executed by the regression tests. Line 12238 was newly added, therefore KATCH used it as a target and eventually executed it. It then attempted to run the program until the end and reached the next line (12240) where it discovered an overflow when reading through the `external` pointer. We have not debugged the exact root cause of the bug ourselves, but we sent an input triggering the crash to the developers, who fixed it shortly.

6. RELATED WORK

Synthesising inputs which cover a target is an essential problem in test generation and debugging and has been addressed through a variety of techniques, including symbolic execution, dependence analysis, iterative relaxation and search-based software testing, among others [2, 15, 21, 36, 40, 41, 44].

---
[5] http://sourceware.org/bugzilla/show_bug.cgi?id=15191

While we borrow ideas from the state of the art in these areas, our approach differs by treating the task as an optimisation problem, where we try to explore paths that minimise the estimated distance to the target using symbolic execution seeded with existing test inputs and enhanced with various heuristics based on program analysis.

We introduced our approach in a workshop paper [28], which phrased the problem in terms of distance minimisation combined with input selection. However, the technique described there was limited to the basic algorithm in §3.3, and did not include any of the advanced program analyses (in particular weakest preconditions for input selection and switching definitions for control dependent branches) which are necessary to make it practical to a wide variety of patch types. Furthermore, the workshop paper only evaluated the approach on three hand-chosen patches, while here we include an extensive evaluation on all the patches written for nineteen programs over a combined period of six years.

Our technique fits within the paradigms of longitudinal and differential program analysis [30, 39], 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 [32] introduces a general framework for using symbolic execution to compute the behavioural characterisation of a program change, and discusses several applications, including regression test generation.

The work most closely related to KATCH is that on directed symbolic execution. Xu and Rothermel introduced directed test suite augmentation [41], in which existing test suites are combined with dynamic symbolic execution to exercise uncovered branches in a patch. The technique is similar to the greedy step in KATCH, without any of our additional analyses.

Researchers have proposed several improvements to this technique: eXpress [36] prunes CFG branches which provably do not lead to the patch; directed symbolic execution [27] introduces call-chain-backward symbolic execution as a guiding technique for symbolic execution; statically-directed test generation [2] uses the size of the target's backward slice reachable from the current state as an estimate for the likelihood of reaching it. Directed incremental symbolic execution [33] is a related technique which improves the efficiency of symbolic execution when having to analyse only the differences between two program versions. It can dynamically prune program paths which exercise the same behaviours in two program versions, and could be combined with KATCH if multiple behaviourally different inputs which cover the patch are desired.

While it is difficult to accurately compare these techniques with KATCH or among each other, we believe that KATCH improves upon previous work in several ways. First, by using the definition switching heuristic, KATCH takes into account more than the currently explored set of paths—and reasoning about unexecuted statements is critical for reaching certain targets. Second, informed path regeneration uses a "surgical" approach to reaching previously infeasible states by making changes to exactly those variables involved in infeasible branch conditions. Third, our evaluation is performed on significantly more patches than in prior work, which gives a better insight into the strengths and limitations of such a technique. Finally, we believe KATCH could be combined with some of these prior approaches, e.g. it could dynamically prune paths that are shown not to lead to the target.

KATCH also shares characteristics with search-based software testing (SBST) [16, 40, 43]. First, our notion of estimated distance is similar to that of *fitness* in SBST. Second, the idea of reusing existing test cases has also been successfully employed in SBST [16, 43]. Future work could try to combine these techniques for the purpose of patch testing.

Research on automatic generation of filters based on vulnerability signatures [4, 10] 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 that lead to that vulnerability. 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 [9]. In the context of automated debugging, execution synthesis [44] and BugRedux [23] attempt to solve a similar problem: generating an input or a path starting from a set of 'waypoints' through which execution has to pass.

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 [22, 34]. While we currently only use simple coverage metrics to guide our analysis, it is possible to combine our approach with such requirements.

7. DISCUSSION AND CONCLUSION

We have presented KATCH, an automated technique for testing software patches. Our approach relies on symbolic execution, augmented by several synergistic heuristics based on static and dynamic program analysis. We have applied KATCH to all the patches written for nineteen programs over a combined period of approximately six years, and have shown that our technique can find bugs and significantly increase patch coverage with only a few minutes per target.

We have learned several lessons from this research. First, it has reminded us that achieving high patch coverage is hard, and that as a result most patches remain untested— e.g. for our benchmarks the manual patch coverage was a modest 31% overall.

Second, it has reinforced our belief that automatic techniques are able to increase patch coverage and find bugs in the process. On average, KATCH was able to increase patch coverage from 31% to 52%, while on the best performing benchmark (`diffutils`), it more than doubled it, from 35% to 73%. In addition, we found fifteen crash bugs in widely-used mature programs.

Finally, it has shown us that the state of the art needs more advances to reach the goal of fully automated testing of real patches: despite the increase in coverage and the bugs found, KATCH was still unable to cover most of the targets in the `binutils` programs. We hope our current results will act as a challenge to other researchers working in this area.

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 2nd edition, 2006.

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

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA'02*.

[4] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE S&P'06*.

[5] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys'11*.

[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*.

[7] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself (invited paper). In *SPIN'05*.

[8] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In *CCS'06*.

[9] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *ASPLOS'09*.

[10] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of Internet worms. In *SOSP'05*.

[11] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel. Staged deployment in Mirage, an integrated software upgrade testing and distribution system. In *SOSP'07*.

[12] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE'05*.

[13] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[14] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *ISSTA'07*.

[15] R. Ferguson and B. Korel. The chaining approach for software test data generation. *TOSEM*, 5(1):63–86, 1996.

[16] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *ICSE'12*.

[17] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC/FSE'99*.

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI'05*.

[19] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS'08*.

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

[21] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *FSE'98*.

[22] R. Gupta, M. Jean, H. Mary, and L. Soffa. Program slicing-based regression testing techniques. *STVR*, 6:83–112, 1996.

[23] W. Jin and A. Orso. Bugredux: Reproducing field failures for in-house debugging. In *ICSE'12*.

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

[25] B. Korel. Automated software test data generation. *TSE*, 16(8):870–879, 1990.

[26] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO'04*.

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

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

[29] P. D. Marinescu and C. Cadar. make test-zesti: A symbolic execution solution for improving regression testing. In *ICSE'12*.

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

[31] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *STVR*, 9(4):263–282, 1999.

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

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

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

[35] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE'05*.

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

[37] N. Tillmann and J. De Halleux. Pex: white box test generation for .NET. In *TAP'08*.

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

[39] J. Winstead and D. Evans. Towards differential program analysis. In *WODA'03*.

[40] Z. Xu, M. B. Cohen, and G. Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *GECCO'10*.

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

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

[43] S. Yoo and M. Harman. Test data regeneration: generating new test data from existing test data. *STVR*, 22(3):171–201, 2012.

[44] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *EuroSys'10*.