`make test-zesti`: A Symbolic Execution Solution for Improving Regression 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 testing is an expensive and time consuming process, often involving the manual creation of comprehensive regression test suites. However, current testing methodologies do not take full advantage of these tests. In this paper, we present a technique for amplifying the effect of existing test suites using a lightweight symbolic execution mechanism, which thoroughly checks all sensitive operations (e.g., pointer dereferences) executed by the test suite for errors, and explores additional paths around sensitive operations. We implemented this technique in a prototype system called ZESTI (Zero-Effort Symbolic Test Improvement), and applied it to three open-source code bases—GNU Coreutils, libdwarf and readelf—where it found 52 previously unknown bugs, many of which are out of reach of standard symbolic execution. Our technique works transparently to the tester, requiring no additional human effort or changes to source code or tests.**

***Keywords*-regression testing; test improvement; symbolic execution**

I. Introduction

Testing currently accounts for a large fraction of the software development life cycle [36] and usually involves writing large numbers of manual tests that exercise various paths, with the objective of maximising a certain coverage metric such as line or branch coverage. This is a tedious process that requires significant effort and a good understanding of the tested system.

As a result, we are witnessing a sustained research effort directed toward developing automatic techniques for generating high-coverage test suites and detecting software errors [4], [8], [12]–[14], [21], [38], [40], with some of these techniques making their way into commercial and open-source tools. However, these techniques do not take advantage of the effort that developers expend on creating and updating the manual regression suites, which we believe could significantly enhance and speed up the testing process.

Our key observation is that well-written manual test suites exercise interesting program features, but often using a limited number of paths and input values. In this paper, we propose a *zero-effort*[1] *symbolic test improvement* technique that amplifies the effect of a regression test suite by checking the program paths explored by the regression suite against *all possible input values*, and also by exploring additional paths that *slightly diverge* from the original execution paths in order to thoroughly check potentially dangerous operations executed by the program.

The paper makes the following main contributions:

1) We show that the effectiveness of standard regression tests can be automatically improved using symbolic execution-based techniques, finding bugs that are often out of reach of regression testing or standard symbolic execution alone. From a tester's perspective, the improved regression suites are executed in exactly the same manner as their non-symbolic counterparts, without the need to configure a symbolic execution engine or decide what symbolic inputs to provide to the program. Our technique works by reasoning about all possible input values on the paths executed by the regression suite and by thoroughly exploring additional paths around sensitive instructions such as dangerous memory accesses.
2) We provide a theoretical and empirical analysis of the sensitivity of our approach to the quality of the test suite, and discuss how the probability of finding a bug varies with the number of test cases being considered.
3) We demonstrate that our approach works well in practice, by implementing it in ZESTI (*Zero-Effort Symbolic Test Improvement*), a prototype based on the KLEE symbolic execution engine [9]. We applied ZESTI to several popular open-source applications, including the `GNU Coreutils` suite, the `libdwarf` library, and the `readelf` utility, and found two previously unknown bugs in `Coreutils` (despite these applications having been comprehensively checked before via symbolic execution [7], [9]), forty in `libdwarf`, and ten in `readelf`, in a manner completely transparent to developers. Furthermore, the inputs generated by ZESTI to reproduce the bugs discovered are *almost well-formed*, i.e. they differ only slightly from the inputs included in the regression suite, making it easier for developers to analyse them.

The rest of the paper is structured as follows. We start by giving a general overview of our technique in §II and discuss the relevant background and related work in §III. We then present our technique in detail in §IV, describe

[1]By *effort* we refer to human effort, not execution overhead.

the main aspects of our implementation in §V, and present our experience using ZESTI in §VI. Finally, we discuss the advantages ZESTI offers, as well as its limitations in §VII, and conclude in §VIII.

II. Overview

The main insight used by ZESTI is that regression test suites exercise *interesting* program paths. Such test suites are often created by the programmers who wrote the application and benefit from deep knowledge of the program logic, or by dedicated QA teams which systematically evaluate the main program features and possible corner cases. Furthermore, regression tests often cover program paths that previously triggered bugs, which are more likely to contain further errors [28], [42]. For instance, while the visible symptoms of the offending bugs are fixed, it can happen that the root cause of the bugs is not; alternatively, slightly different executions could still trigger the same bug.

A common way to measure the quality of a test suite is code coverage. Testing methodologies often require creating test suites that achieve a certain level of line or branch coverage, and many projects contain relatively high-coverage test suites: for instance, most applications that we tested in our experimental evaluation have manual test suites reaching over 60% line coverage.

Unfortunately, despite the effort invested in creating these manual regression suites, bugs still remain undetected in the code covered by the test inputs. First of all, note that line coverage can be misleading for quantifying the confidence in a system for two important reasons. First, executing an operation may or may not cause a violation depending on its arguments. For example accessing the *i-th* element of a vector is safe when i is within vector bounds but causes an error otherwise. Line coverage, however, considers the operation tested as soon as it is executed once. Second, code behaviour depends on the path used to reach it; an instruction can operate correctly when reached along one path but cause a violation along a slightly different path. These caveats also apply to other coverage metrics, such as branch coverage.

In this paper, we propose to augment regression suites by using symbolic execution [24] to (1) analyse instruction safety against all inputs that *could* exercise the instruction along the same paths (§IV-A) and (2) carefully choose and explore *slightly divergent* paths from those executed by the regression suite (§IV-B). Compared to standard regression suites, our approach tests the program on all possible inputs on the paths exercised by the regression suite and on a large number of neighbouring paths, without any additional developer effort. Compared to standard symbolic execution, the approach takes advantage of the effort spent creating these regression suites, to quickly guide symbolic execution along paths that exercise interesting program behaviours.

III. Background and Related Work

Symbolic execution is a technique that has received much attention in recent years in the context of software testing [10] due to its ability to automatically explore multiple program paths and reason about the program's behaviour along each of them. Tools such as KLEE [9], SAGE [17], JPF-SE [1], BitBlaze [33] and Pex [37] are just some of the symbolic execution engines currently used successfully in academia and in industry.

Intuitively, symbolic execution (SE) works by systematically exploring all possible program executions and dynamically checking the safety of dangerous operations. SE replaces regular program inputs with *symbolic* variables that initially represent any possible value. Whenever the program executes a conditional branch instruction that depends on symbolic data, the possibility of following each branch is analysed and execution is forked for each *feasible branch*. To enable this analysis, symbolic execution maintains for each execution path a set of conditions which characterise the class of all inputs that drive program execution along that path. At any time, the *path conditions* can be solved to provide a concrete input that exercises that path natively, making it easy to reproduce, report and analyse an execution of interest. SE also analyses all potentially dangerous operations as they are executed, verifying their safety for any input from the current input class. For example, a division is safe if and only if the denominator cannot be zero given the current path conditions.

Unfortunately, the number of execution paths in real programs often increases exponentially with the number of branches in the code, which may lead symbolic execution to miss important program paths. One solution is to employ sound program analysis techniques to reduce the complexity of the exploration, e.g., as in [6] or [15]. An orthogonal solution is to limit or prioritise the symbolic program exploration using different heuristics. For example, directed symbolic execution methods [3], [11] use techniques such as static analysis to find instructions or paths of interest which are then used to guide the symbolic exploration.

Concolic testing [16], [32] starts from the path executed by a concrete input, and then explores different program paths by systematically flipping the truth value of the branch conditions collected on that path. Previous research has shown that the coverage and bug-finding abilities of concolic testing can be improved by combining it with random testing [27] or with well-formed inputs [17], and the effectiveness of fault-localization can be increased by aiming to maximize the similarity with the path constraints of faulty executions [2].

Combining concolic execution with manual test suites was first proposed in [23], where it was augmented by assertion hoisting in order to increase the number of bug checks, and then explored in [39], in which it was compared against a genetic algorithm test augmentation technique. Our approach extends previous work by proposing a technique that explores paths around potentially dangerous instructions executed by the regression suite, by providing an analysis of the sensitivity of this approach to the quality of the test suite, and by presenting a thorough evaluation on real and complete regression suites of several popular applications.

The idea of augmenting concrete executions with the ability to reason symbolically about potential violations was first proposed in [25], which introduces a technique that keeps track of lower and upper bounds of integer variables, and of the `NUL` character in strings. Based on this information, it can flag bugs such as buffer overflows and incorrect uses of `libc` string functions. The technique can only reason about limited types of constraints, and does not explore any additional paths.

Research on improving regression testing generally falls under four main categories: (1) finding redundant tests [19], (2) ordering tests for finding defects earlier [34], (3) selectively running only relevant tests on new program versions [30] and (4) enhancing a system's test suite as the system evolves [5], [18], [31], [35], [41]. The first three topics are largely orthogonal and can be combined with our technique. The state-of-the-art for the latter combines control- and data-dependence chain analysis and symbolic execution to identify tests that are likely to exercise the effects of changes to a program. Making this approach tractable requires a reasonably small set of differences between program versions and a depth-bounded analysis on dependence chains. While our approach could be used for test augmentation, we see ZESTI primarily as a bug-finding technique that can increase the effectiveness of regression suites by combining them with symbolic execution, following the manner in which dynamic execution tools such as Valgrind are often integrated with existing test suites.

IV. Zero-Effort Symbolic Test Improvement

This section describes the two main techniques used by ZESTI: improving regression suites with additional symbolic checks (§IV-A), and exploring additional paths around sensitive operations (§IV-B).

A. Thoroughly Testing Sensitive Operations

A standard regression test suite consists of multiple tests, each being an (input, expected output) pair. The test harness iterates through the tests and runs for each of them the target program with the given input and collects its output. ZESTI hooks into this process by interposing between the testing script and the tested program, gaining complete control over the system's execution.

Similarly to [23], ZESTI replaces the program input with symbolic values and at the same time remembers the concrete input, which is used to drive program execution whenever a branch is encountered. While executing the program, path conditions are gathered and used to verify potentially buggy operations. For example, whenever the program accesses a symbolic memory location, ZESTI checks that the operation is safe for all inputs that satisfy the current path condition.

```
1: int v[100];
2: void f(int x) {
3:    if (x > 99)
4:       x = 99;
5:    v[x] = 0;
6: }
```

Figure 1. Code snippet showcasing a bug missed by a test suite with 100% code coverage, e.g. x=50, x=100.

Consider the snippet of code in Figure 1. Function `f` contains a bug: it accesses an invalid memory location when passed a negative argument. A test suite might call this function with different arguments and verify its behaviour, attempting to maximise a certain metric, e.g., line coverage. It can be easily noticed that choosing one value greater than 99 and one smaller than or equal to 99 exercises all instructions, branches and paths without necessarily finding the bug. On the other hand, our approach finds the bug whenever the function argument is smaller than 100: for such values, symbolic execution gathers the path constraint $x \leq 99$ on line 3, and then on line 5 checks whether there are any values for x than can overflow the buffer v. More exactly, ZESTI checks whether the formula $x \leq 99 \Rightarrow (x \geq 0 \wedge x \leq 99)$ is valid and immediately finds a counterexample in the form of a negative integer assignment to x.

In order to be accepted by software developers, we strongly believe that ZESTI needs to work transparently. We envision ZESTI being used in a similar way in which memory debuggers such as Valgrind [29] or Purify [20] are employed today in conjunction with test suites to provide stronger guarantees. For example, many open-source programs provide a simple way to integrate Valgrind into their regression test frameworks, with the user simply having to type "`make test-valgrind`" to execute the regression suite under Valgrind. We hope ZESTI will be used in a similar way, by simply typing a command like "`make test-zesti`", as suggested in our paper title.

In other words, running an existing regression test suite under ZESTI should happen without user intervention. To accommodate all testing frameworks, ZESTI treats both the tests and the testing script as black boxes. It functions by renaming the original program and replacing it with a script that invokes the ZESTI interpreter, passing as arguments the original program and any command line arguments.[2]

[2]Because ZESTI is an extension of the KLEE symbolic execution engine, which operates on LLVM bitcode [26], users need to compile their code to LLVM in order to use ZESTI. However, this is not a fundamental limitation of our approach, which could be integrated within a symbolic execution framework that works directly on binaries.

```
Inputs: $MaxDist$, the maximum distance to search,
        $S$, the set of sensitive instructions,
        $P$, the set of divergence points
        $f$, the distance estimation function

1: for $D$ = 1 to $MaxDist$
2:     for sensitive instructions $I \in S$
3:         if $\exists$ divergence point $J \in P$
              at distance $D$ from $I$
4:         symbolically execute program starting
           from $J$, without restriction to a
           single path, with depth bound $f(D)$
```

Figure 2. Algorithm used by ZESTI to explore additional paths.

ZESTI automatically detects several input classes, namely command-line arguments and files opened for reading, and treats them as sources of symbolic data. We found these two sources sufficient for our benchmarks; however, adding additional input sources is relatively straightforward.

The main downside of this approach is execution overhead. In particular, there are two main sources of overhead: first, the overhead of interpreting LLVM code. Second, the constraint solver overhead: however, note that unlike in regular symbolic execution, the constraint solver is invoked in ZESTI only to check sensitive operations.

B. Exploring Additional Paths Around Sensitive Operations

The version of ZESTI described thus far has the disadvantage of being highly dependent on the thoroughness of the regression test suite. While a quality test suite is expected to test all program features, it is likely that not all corner cases are taken into account. Our analysis of `Coreutils` (§VI), a mature set of applications with a high quality test suite, showed that only one out of the ten bugs previously found via symbolic execution could be detected by the version of ZESTI described so far. As a result, we extended ZESTI to explore paths that *slightly diverge* from those executed by the regression suite, according to the likelihood they could trigger a bug.

To mitigate the path explosion problem, ZESTI carefully chooses divergent paths via two mechanisms: (1) it only diverges close to *sensitive instructions*, i.e instructions that *might* contain a bug, and (2) it chooses the divergence points in order of increasing distance from the sensitive instruction. The key idea behind this approach is to exercise sensitive instructions on slightly different paths, with the goal of triggering a bug if the respective instructions contain one. Choosing a close divergence point ensures that only a small effort is needed to reach the same instruction again.

ZESTI identifies sensitive instructions dynamically. As it executes the concrete program path, it keeps track of all instructions that might cause an error on alternative executions. We consider two types of sensitive instructions: memory accesses and divisions. We treat all pointer dereferences as sensitive, while for divisions we only consider those with a symbolic denominator as sensitive. At the LLVM level, ZESTI treats as sensitive all memory accesses to symbolic addresses, as well as those (concrete or symbolic) memory accesses preceded by a `GetElementPtr` instruction, and all division and modulo operations with symbolic denominators. While we currently track only sensitive memory accesses and divisions, we could also extend the technique to other types of sensitive operations, such as assertions.

To comprehensively exercise the sensitive instructions with different inputs, ZESTI tries to follow alternative execution paths that reach these instructions. To this purpose, it identifies all points along the concrete path where execution can diverge, i.e. the branches depending on symbolic input. ZESTI then prioritises the divergence points in increasing order of *distance* from sensitive instructions and uses them as starting points for symbolic execution. Figure 2 outlines the strategy used by ZESTI. Line 1 goes through distances from 1 to a user-specified maximum and line 2 iterates through all sensitive instructions. If any divergence point is found at the current distance from the current instruction, it is used to start a depth-bounded symbolic execution run, with bound $f(D)$. The function f should be a function that closely overestimates the distance between the divergence point and the sensitive instruction on an alternative path. A function which underestimates this distance will give an SE depth bound too small to reach the sensitive instruction, while a function that largely overestimates it would needlessly increase ZESTI's overhead. (However, note that not all additional paths explored by ZESTI are guaranteed to reach the sensitive instruction.) We empirically found that a linear function works well, and in our experiments we used $f(D) = 2 * D$.

As an optimisation, line 2 of the algorithm considers sensitive instructions in decreasing order of distance from the program start. This favours the exploration of deeper states first, on the premises that (1) deeper states are more interesting because they contain the functionality exercised by the test suite as opposed to the shallow states that are often related to command-line parsing or input validation, and (2) standard symbolic execution is less likely to be able to reach those states in reasonable time due to path explosion.

Intuitively, the metric used by ZESTI to measure the distance between two execution points needs to estimate the effort required by symbolic execution to reach one point from the other. To this end, ZESTI defines the distance between two instructions as the number of branches between them where inputs could allow execution to follow either side of the branch. This metric captures the number of points where the program could have taken a different path (and which ZESTI could explore), and is inherently insensitive to large blocks of code that use only concrete data.

In practice, the optimal maximum distance (`MaxDist` in Figure 2) for which to run ZESTI is hard to determine. Using

```
Depth   Code                       InstrType
--------------------------------------------
        int v[100];
        void f(int x) {
0         if (x > 99)              D
            x = 99;
1         v[x] = 0;                S
        }
```

Figure 3. Code snippet showcasing an execution generated by an input $x > 99$, annotated by ZESTI. The `Depth` column records the distance from the start of the execution, and the `InstrType` column keeps track of divergence points (`D`) and sensitive instructions (`S`).

```
Depth   Code                       InstrType
--------------------------------------------
        int v[100];
        void f(int x) {
0         if (x > 99) {            D1
1           if (x > 199)           D2
              return;
            x = 99;
          }
2         v[x] = 0;                S
        }
```

Figure 4. Code snippet showcasing an execution generated by an input $99 < x \leq 199$, annotated by ZESTI. The `Depth` and `InstrType` columns have the same meaning as in Figure 3.

a small value may miss bugs, while using a large value may be too time-consuming and leave no time to execute the rest of the tests within the allocated time budget. Our approach to solve this problem is to allocate a certain time budget to the entire testing process and use an iterative deepening approach: conceptually, all the tests are first executed without exploring any additional paths, then up to distance 1, 2, 3, etc., until the time budget expires.

To illustrate ZESTI's exploration of additional paths, consider the code in Figure 1. In the previous section we showed how ZESTI finds the bug starting from a test that calls function `f` with an argument smaller than 100. We now show how it can find the bug for any argument value. For values smaller than 100, the previous analysis applies and the bug is found without having to explore divergent paths. Therefore, we only discuss arguments greater than or equal to 100. Figure 3 shows the same code, annotated by ZESTI, when executed using such an argument. While running the function, ZESTI records all the sensitive instructions (`S`), and divergence points (`D`) being executed (`InstrType` field), and computes their distance from the start of the execution (`Depth` field).

After running the entire function, ZESTI looks for instructions labelled as sensitive located at distance 1 after a divergence point (i.e., the difference between their `Depth` fields is 1), and finds instruction `v[x] = 0` with corresponding divergence point `if (x > 99)`. These steps correspond to lines 2 and 3 of Figure 2. ZESTI then starts bounded symbolic execution from `D` (line 4 of Figure 2). The new path discovered corresponds to an input that makes the code take the (empty) `else` branch at `D`, i.e. a value smaller than 100. On this path `x` is no longer set to 99 but is used directly to index `v`. When executing the sensitive instruction `v[x] = 0`, ZESTI checks whether a violation can occur based on the current path condition, and finds that a negative function argument causes a memory violation.

To further illustrate ZESTI's algorithm, we consider the slightly more complicated code snippet in Figure 4. The code contains an additional `if` statement that creates a new divergence point `D2`. Assuming a test input between 100 and 199, the sensitive instruction is at distance 1 from divergence point `D2` and at distance 2 from `D1`. Therefore, ZESTI first considers `D2`, and explores its `then` path, which does not trigger the bug. Going further, it finds `D1` which leads to the bug as in the previous example.

C. Improving Efficiency by Discarding Test Cases

An interesting question is how sensitive ZESTI is to the program test suite. The time in which ZESTI finds a bug depends on three main factors: the number of tests that are run, the percentage of them that expose the bug, and the minimum distance at which the bug is found.

As discussed above, because the distance at which a certain test case exposes the bug is difficult to predict, ZESTI first checks the concrete execution path and then uses an iterative deepening approach to check divergent paths. Under this strategy, the only other parameter that ZESTI can vary is the number of test cases that are run. In the rest of this section, we provide a theoretical estimate of the probability of finding a bug if ZESTI runs only a randomly chosen fraction of the test suite.

Creating a sub-test suite out of the initial test suite by randomly picking tests is an instance of the *urn model without replacement* [22], i.e. the marbles (tests) are not replaced in the urn (initial test suite) once picked. Consider that the urn model has the following parameters: N – the total number of tests, m – the number of tests which expose the bug at the minimum distance, and k – the number of tests picked. The probability distribution which characterises the number of successes (i.e. tests which find the bug at the minimum distance) in the sequence of k picks is called the *hypergeometric distribution* [22].

In terms of this model, we are interested in the probability of having at least one success, which is $1 - P(failure)$, the probability of having only failures:

$$P(success) = 1 - P(failure) = 1 - \binom{N-m}{k} \Big/ \binom{N}{k}$$

where the fraction denominator represents the total number of possible test combinations and the numerator represents the number of combinations which contain zero successes.

Figure 5 plots the probability of finding a bug using a subset of a hypothetical initial test suite of 100 test cases

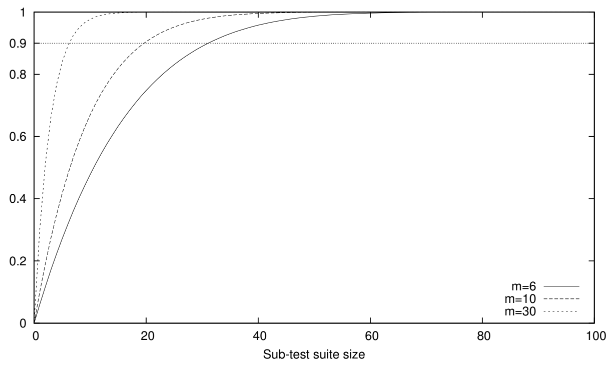

Figure 5. Probability to find a bug using a randomly picked fraction of an initial test suite of 100 test cases. The three lines show the probability considering that 6%, 10% and respectively 30% of the initial tests find the bug.

for three fractions of tests exposing the bug: 6%, 10% and 30%, which are representative for the programs that we analysed with ZESTI (see §VI). As this graph shows, it is possible to discard a large fraction of the test suite while still finding the bug with high probability. For example, for a test suite of size 100, in order to find the bug with at least 90% probability, it is enough to run only 7 (when *m*=30%), 20 (when *m*=10%), and 32 tests (when *m*=6%). If the minimum distance at which the bug is found is relatively large, discarding a large number of tests can have a big positive impact on ZESTI's performance, without significantly lowering the probability of finding the bug. In Section VI-C we show that our analysis holds in practice by examining the test suite characteristics of real programs.

V. IMPLEMENTATION

ZESTI is integrated in the KLEE symbolic execution engine [9]; a user can choose whether to run KLEE or ZESTI via command line switches. When enabled, ZESTI intercepts all calls that create symbolic data, (e.g., read from a file), and records the concrete value of the variables in a shadow data structure. ZESTI also intercepts all calls made to the STP constraint solver, via a custom *concretizing* module inserted in KLEE's solver chain between the front-end and the query optimisers. When enabled, this module replaces all symbolic variables in a query with their concrete values and evaluates the resulting concrete expression, obtaining a value that is then returned directly back to KLEE. This implementation allows enabling and disabling symbolic execution by disabling and respectively enabling ZESTI's concretizing module. The module is always disabled before executing a sensitive operation such as a memory access and re-enabled afterwards. This permits checking sensitive operations symbolically while executing the rest of the program concretely.

In order to explore paths around sensitive instructions, ZESTI associates with each program state that is not on the concrete path a *time-to-live (TTL)* value which keeps track of how long this state continues to be executed before it is suspended. This mechanism allows executing states in any order and guarantees execution for the exact desired distance. The TTL uses the same metric used to measure distances between program states, i.e. symbolic branch count. It is initialised with the distance for which the state has to be executed, and decremented whenever the state is forked as a result of a symbolic branch condition.

ZESTI also implements its own state prioritization algorithm based on a breadth-first traversal of the state space, consistent with the distance metric used. The algorithm is implemented as a *searcher*, a pluggable abstraction used by KLEE to encapsulate the prioritization logic. This approach decouples the search algorithm from the symbolic execution functionality and allows updating or replacing the implementation with ease.

VI. EXPERIMENTAL EVALUATION

This section covers the results obtained with ZESTI, describing our benchmarks and methodology (§VI-A), bugs found (§VI-B), and quantifying the test improvements and overhead of using ZESTI (§VI-C).

A. Benchmarks

To evaluate ZESTI, we used three different software suites:

1) **`GNU Coreutils 6.10`**, a suite of commonly-used UNIX utilities such as `ls`, `cat` and `cp`. `Coreutils` consists of a total of 89 individual programs and has a comprehensive regression test suite totalling 393 individual tests obtaining overall 67.7% line coverage. We used the older 6.10 version in order to facilitate the comparison against KLEE, which was previously used to comprehensively check this version of `Coreutils` [9]. The largest `Coreutils` program (`ls`) has 1429 effective lines of code (ELOC) but also uses part of a monolithic library shared by all the utilities, making it hard to compute an accurate line count. We therefore employed the same approach used by KLEE's authors, of computing the number of LLVM instructions after compiler optimisations are applied (especially the dead code elimination pass). This yields 20,700 instructions for `ls`.[3]

2) **`libdwarf 20110612`**, a popular open-source library for inspecting DWARF debug information in object files. `libdwarf` has 13,585 ELOC as reported by gcov and 31,547 LLVM instructions, as reported by KLEE. Its test suite consists of two parts: 88 manually created tests and a larger number of automatically-generated tests obtained by exhaustively mixing common command-line arguments and input files, achieving in total 68.6% line coverage.

[3]Line count and coverage information was obtained using `gcov` 4.4.3 and `LLVM` 2.9. Numbers can vary between different versions.

3) **`readelf 2.21.53`**, a component of `GNU binutils` for examining ELF object files, included in most Linux distributions. `readelf` has 9,938 ELOC and 30,070 LLVM instructions, and comes with a small test suite of only seven tests obtaining 24% line coverage. One reason we included this benchmark was to see how ZESTI performs with a weaker regression suite. The other was that both `libdwarf` and `readelf` need large inputs (executable files), which would make a pure symbolic execution choke. For example, executing `libdwarf` using KLEE and a relatively small, 512 byte input file consumes all available memory on our test machine within a few tens of minutes.

To test these programs, we imposed a per-test time limit dependent on program complexity: we chose 15 minutes for the `Coreutils` programs and 30 minutes for `libdwarf` and `readelf`. For `libdwarf`, we used the 88 manual tests and 12 of the automatically-generated ones. We ran all `libdwarf` experiments on a 64bit Fedora 16 Xeon E3-1280 machine with 16GB of RAM, while the rest were performed on a 64bit Ubuntu 10.04 i5-650 machine with 8GB of RAM.

B. Bugs Found

ZESTI found a total of 58 bugs, out of which 52 were previously unknown. The new bugs were reported to the maintainers and most of them have already been fixed by the time of this writing. Table I shows a summary of the bugs found by ZESTI, along with the distance from the concrete path and the depth at which they were found. We compute the depth as the number of visited symbolic branches from the program start where both sides could be explored, as this is a rough estimation of the effort required by standard symbolic execution to find the bug. If the same bug is discovered by two or more test cases we report the minimum distance and for equal distances the minimum depth. Both the minimum distance and depth are influenced by program inputs; it may be possible to reach the bugs by traversing fewer symbolic branches when using other inputs.

We describe below three representative errors found by ZESTI, and then compare its bug-finding ability against standard symbolic execution.

Table I
BUGS FOUND BY ZESTI ALONG WITH THE DISTANCE (FROM THE CONCRETE TEST PATH) AND THE DEPTH (FROM THE PROGRAM START) AT WHICH THE BUG WAS FOUND. NEW BUGS ARE IN BOLD.

Bug no.	Location	Distance	Min Depth
`Coreutils`			
1	**cut.c:267**	**0**	**65**
2	**printf.c:188**	**1**	**9**
3	seq.c:215	1	7
4	paste.c:107	1	8
5	mkdir.c:192	6	9
6	mknod.c:169	8	12
7	mkfifo.c:117	6	10
8	md5sum.c:213	10	45
`libdwarf`			
9	**dwarf_form.c:458**	**2**	**491**
10	**dwarf_form.c:503**	**0**	**1229**
11	**dwarf_form.c:525**	**0**	**490**
12	**dwarf_elf_access.c:663**	**0**	**382**
13	**dwarf_elf_access.c:664**	**0**	**383**
14	**dwarf_arange.c:160**	**0**	**319**
15	**dwarf_arange.c:179**	**0**	**321**
16	**dwarf_util.c:90**	**0**	**746**
17	**dwarf_util.c:396**	**0**	**923**
18	**dwarf_elf_access.c:640**	**0**	**495**
19	**dwarf_print_lines.c:385**	**0**	**514**
20	**dwarf_global.c:305**	**0**	**2057**
21	**dwarf_global.c:239**	**0**	**1508**
22	**dwarf_global.c:267**	**2**	**400**
23	**dwarf_leb.c:58**	**0**	**396**
24	**dwarf_leb.c:62**	**1**	**650**
25	**dwarf_leb.c:69**	**1**	**650**
26	**dwarf_leb.c:128**	**1**	**650**
27	**esb.c:117**	**0**	**1248**
28	**print_die.c:1523**	**0**	**1292**
29	**dwarf_util.c:116**	**0**	**488**
30	**dwarf_util.c:363**	**0**	**1248**
31	**dwarf_util.c:418**	**0**	**498**
32	**dwarf_query.c:325**	**0**	**648**
33	**dwarf_abbrev.c:119**	**0**	**543**
34	**dwarf_frame2.c:936**	**1**	**376**
35	**dwarf_frame2.c:948**	**0**	**389**
36-48	**dwarf_line.c:*[4]**	*	*
`readelf`			
49	**readelf.c:5020**	**0**	**134**
50	**readelf.c:10140**	**0**	**285**
51	**readelf.c:10600**	**0**	**73**
52	**readelf.c:10607**	**5**	**51**
53	**dwarf.c:182**	**0**	**277**
54	**dwarf.c:549**	**0**	**276**
55	**dwarf.c:2596**	**0**	**585**
56	**elfcomm.c:69**	**0**	**287**
57	**elfcomm.c:142**	**0**	**258**
58	**elfcomm.c:149**	**0**	**261**

***`cut`* case study:** The bug found in the `cut` utility is a memory access violation. The test leading to its discovery uses the command line arguments `-c3-5,6- --output-d=: file.inp`. The `-c` argument specifies two ranges, from the 3rd to the 5th character and from the 6th character to the end of the line. Internally, `cut` allocates a buffer that is later indexed by the range endpoints. Its size is computed as the maximum of the right endpoints across all ranges. However, in this case, the ranges unbounded to the right are incorrectly not considered in the computation. Therefore the value 6 is used to index a (zero-based) vector of only 6 elements. However, because the `cut` implementation uses a bitvector, allocations are inherently done in chunks of 8 elements and the bug is not triggered by the test input (and thus a tool such as Valgrind could not find it). However, ZESTI detects the problem by deriving a new input which triggers the bug, namely `-c3-5,8- --output-d=: file.inp`.

***`libdwarf`* case study:** One of the bugs found in `libdwarf` is a division by zero, caused by improper handling of debug

[4]Bugs were found at 13 different locations in `dwarf_line.c`. For brevity we omit the details.

Table II
A ONE BYTE CORRUPTION AT OFFSET 0X1073 IN A `LIBDWARF` TEST FILE, WHICH CAUSES A DIVISION BY ZERO.

Offset	Original	Buggy
`0000`	`7F 45 4C 46`	`7F 45 4C 46`
...	...	...
`1070`	`00 00 00` **04**	`00 00 00` **00**
...	...	...
`2024`	`69 74 00`	`69 74 00`

information data. Before reading the debug *aranges* section, `libdwarf` computes the size of each entry by looking at two fields in the executable file: the address size and the segment size. The entry size is computed using the formula $entry\_size = 2 * address\_size + segment\_size$. A check is then made to ensure that the section size is a multiple of the entry size via a modulo operation, which causes an exception when the entry size equals zero.

Table II shows the input generated by ZESTI by changing one byte in the original test file. The byte corresponds to the address size, which is changed from 4 to 0 (the segment size is already 0). The new file causes the division by zero when passed to `libdwarf`. One advantage of ZESTI over standard symbolic execution is that it can generate "almost well-formed" inputs. While symbolic execution can only use the current path constrains to generate an input, leaving all unconstrained data to a default value, ZESTI creates an input that matches as close as possible the test data, while still reproducing the bug. The feedback to our bug reports indicates that this approach creates inputs that are easier to understand by programmers.

***`printf`* case study:** ZESTI found a previously unknown bug in the `printf` program, a utility that prints formatted text in a similar fashion to the `printf libc` function. The bug was found at distance 1, i.e., ZESTI had to flip the outcome of one branch in order to trigger it. The bug resides in a program feature that interprets a character as its integer ASCII code if preceded by a single or double quote. The implementation incorrectly assumes that all quotes are followed by at least one character; when a lone quote is provided as input, an off-by-one memory access is performed. ZESTI infers from the `printf %c x` test, the input `printf %d '`, which triggers the bug.

Comparison with standard symbolic execution: In terms of bug-finding capabilities, ZESTI and KLEE enjoy different advantages. On the one hand, ZESTI is able to avoid certain scalability problems that symbolic execution is facing, by using the paths executed by the regression suite to reach interesting program states. For example, ZESTI was able to find forty bugs in `libdwarf` and ten in `readelf`, while KLEE was not able to find any of them, because it 'got lost' in the large program state space, ending up consuming all available memory on our test machine. The large depth

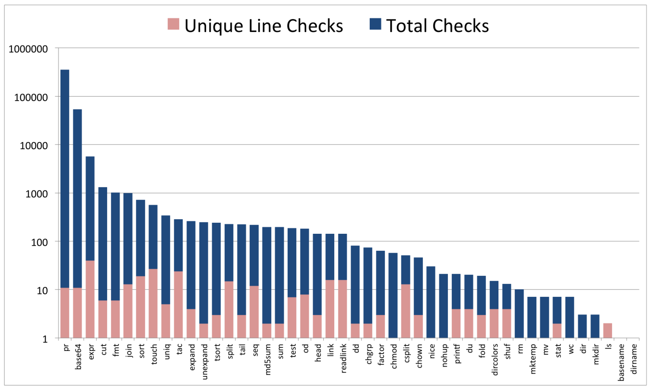

Figure 6. Number of unique (by line of code) and total checks performed by ZESTI on `Coreutils` 6.10.

at which the `libdwarf` and `readelf` bugs are found in the symbolic state tree (*Min Depth* column in Table I) shows that symbolic execution needs to search through a significantly larger number of states. For example, to find a bug at depth 100 requires searching through roughly 2^{90} times more states than it does for a bug at depth 10.

On the other hand, four of the bugs found by KLEE were not detected by ZESTI, showing its limitations. One of the bugs, found in the `tac` utility, is only triggered when providing more than one input file to the program. Because none of the tests do so, the buggy code is never executed in the inconsistent state. The two bugs found by KLEE in `ptx` are missed because the regression suite does not contain any tests for this program. Finally, the bug in the `pr` utility was not found due to the highly solver-intensive test inputs, which were consuming all the allocated time budget on the concrete path, not allowing ZESTI to explore beyond it in the allocated time budget.

C. Symbolic Bug Checks and Performance Overhead

Symbolic bug checks: One measure of ZESTI's effectiveness is the number of symbolic checks (in our case memory access checks) made when running a regression suite. Figure 6 shows the number of total and unique checks performed for each program in the `Coreutils` suite when running ZESTI on the regression suite with distance 0 (i.e., with no additional paths explored) and a timeout of two minutes per program execution. Uniqueness was determined solely through the line of code that triggers the check.

Figure 6 shows 46 bars, one for each `Coreutils` application in which ZESTI performed symbolic checks while running the regression suite. The rest of the `Coreutils` programs do not provide any opportunities for such checks because they either are too simple (e.g., `yes`), do not take user input, or do not use it to access memory, (e.g., `id`, `uname`). This does not represent a limitation of ZESTI but instead shows that not all programs are susceptible to memory access bugs.

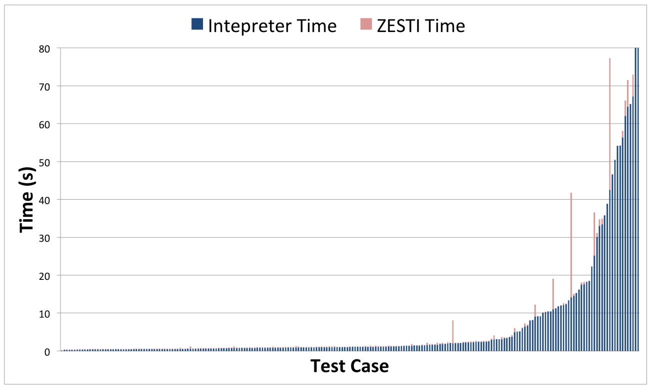

Figure 7. ZESTI execution overhead compared to KLEE as an interpreter, when run with distance 0 on the `Coreutils` regression suite.

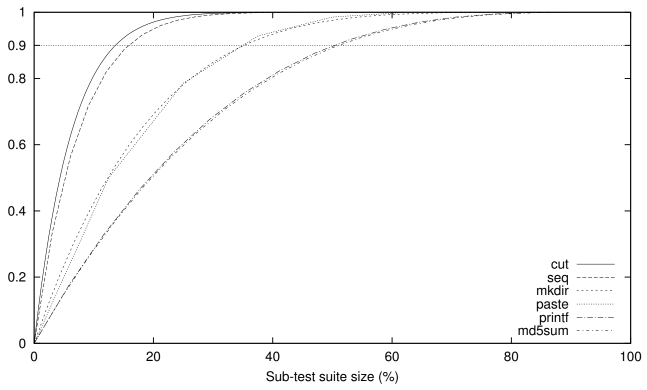

Figure 8. Probability to find the `Coreutils` bugs at the minimum distance, relative to the size of a randomly chosen sub-test suite.

Overhead of* ZESTI*'s checks: Under the same setup, we also measured the time taken by ZESTI to run each test in the regression suite. To compute ZESTI's overhead, we use as baseline KLEE as an interpreter only, i.e. without any symbolic data. Because no symbolic data is introduced, KLEE uses its system call models, object management system and the same internal program representation as in symbolic execution mode but follows only one execution path and does not use the constraint solver.

To eliminate potential inconsistencies, we only consider tests that complete successfully, as reported by the regression suite. This eliminates 21 tests that result in ZESTI timeouts and a small number of early program exits due to imperfections in `uClibc` or KLEE's models, which would otherwise add noise to our experiments.

The results are presented in Figure 7, which shows one pair of bars for each program execution: one for the time taken by the interpreter, and one for the time taken by ZESTI. The times are sorted by interpreter time. The last two tests, not completely shown, take 250 seconds to terminate under the interpreter and have less than 1% overhead under ZESTI. We see that for most tests, the execution times are virtually identical for KLEE and ZESTI. However, there are several executions for which ZESTI takes significantly more time, due to the constraint solver queries that it issues while making the symbolic checks. Finally, note that the interpreter time adds significant overhead on top of native execution (which for `Coreutils` usually takes only milliseconds per program execution), and one way to improve ZESTI's performance is to speed-up the interpreter (which in KLEE is not optimised, because in standard symbolic execution it is rarely a bottleneck).

Effect of discarding test cases: Table III shows the size of the test suite for each application in `Coreutils` for which ZESTI found a bug (`#T`), and the distance distribution for the bugs, across all available tests for each program (`D0-D12`). The `Not found` value corresponds to not finding the bug in 15 minutes (60 minutes for `md5sum`).

Based on the information in Table III and using the formula presented in Section IV-C, we plotted in Figure 8 the probability of finding the bug at the minimum distance for each of these applications, relative to the size of a randomly chosen sub-test suite. It can be noticed that the worst scenarios correspond to the `printf` and `md5sum` programs, where more than half of the tests are needed to have at least 90% confidence in finding the bug. For the rest of the programs, a confidence of at least 90% can be achieved by using roughly one third (or less) of the tests. This indicates that in practice, it might be possible to improve ZESTI's efficiency—without significantly affecting the probability of finding a bug—by randomly discarding a large part of the test suite. `libdwarf`'s test suite corroborates these results, while `readelf` has a test suite too small to be considered for this analysis.

VII. Discussion and Future Work

The ultimate goal of our project is to make ZESTI accessible to testers through a simple, standardised interface. Most systems already include a regression test suite, usually invoked from the command line via a `make check` or `make test` command. Some systems also allow running the regression suite through a memory debugger such as Valgrind, using a simple command such as `make test-valgrind`, in order to catch invalid memory operations which do not result in observable errors. We envision exposing ZESTI through a similar command, e.g. `make test-zesti`, which would enable all the additional checks made by it.

The approach implemented by ZESTI has several advantages: (1) it does not require changes to the program source code or to the regression tests, as ZESTI is interposed transparently between the test harness and the actual program; (2) it takes advantage of the effort put in the original test cases, as they are reused to drive symbolic execution under ZESTI; and (3) for each bug found, an input that reproduces the bug is generated; furthermore, to help developers understand the bug, this input is kept as similar as possible to the original test input.

Table III
Bug distance distribution for the bugs found by zesti in `Coreutils`. The minimum distance at which the bug is found for each program is in bold. The **Not found** value corresponds to not finding the bug in 15 minutes (60 minutes for `md5sum`).

App	#T	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	Not found
cut	163	**9.2%**	–	–	–	–	–	–	–	–	–	–	–	–	90.8%
printf	17	–	**17.6%**	5.9%	5.9%	–	–	–	–	–	17.6%	–	–	–	53.0%
md5sum	22	–	–	–	–	–	–	–	–	–	–	**13.6%**	–	–	86.4%
mkdir	44	–	–	–	–	–	–	**11.3%**	–	4.5%	4.5%	–	–	–	79.7%
mknod	1	–	–	–	–	–	–	–	–	**100%**	–	–	–	–	–
mkfifo	1	–	–	–	–	–	–	**100%**	–	–	–	–	–	–	–
paste	8	–	**50.0%**	–	–	–	–	–	–	–	–	–	50.0%	–	–
seq	33	–	**33.3%**	–	3.0%	9.1%	3.0%	9.1%	–	3.0%	–	18.2%	–	6.1%	15.2%

The main disadvantage of this approach is that it can take significantly more time than natively executing the regression tests. However, our empirical analysis showed that a good regression suite allows finding bugs close to the concrete execution path, thus minimising the time spent symbolically executing the program. Furthermore, ZESTI can be tuned to specific time budgets through various configurable settings which limit the exploration via timeouts (per-instruction, per-solver query, per-branch from the concrete path) or by disallowing execution beyond a certain distance. Finally, if necessary, developers can only run a part of the test suite under ZESTI, often without significantly lowering the probability of finding bugs.

One of the problems of symbolic execution is that it can get stuck in uninteresting parts of the code, such as input parsing code, and therefore miss interesting "deep paths." ZESTI solves this problem by first executing the entire program along the paths from the regression suite, and then exploring additional branches symbolically, in increasing distance from sensitive instructions and the program end.

One problem that we observed in the `pr` utility from the `Coreutils` suite is a very expensive—in terms of symbolic checks—concrete path. This prevents ZESTI from exploring paths which diverge from the test suite in the given time budget. In the future, we plan to incorporate in ZESTI techniques for adaptively skipping checks, effectively allowing the tester to trade checks at a lower depth for checks at a higher depth.

Unlike symbolic execution, ZESTI eliminates the guesswork involved in setting up symbolic data. In particular, choosing the appropriate number and size of symbolic inputs is non-trivial: on the one hand, small inputs may miss bugs, while on the other hand large inputs can significantly increase execution overhead, by generating very expensive constraint solving queries, or by causing symbolic execution to spend most of its time in non-interesting parts of code. While analysing the two `Coreutils` bugs detected by ZESTI but missed by KLEE, we found that carefully tuning the symbolic input size allows standard symbolic execution to find them. Surprisingly, one of the bugs can be found only with larger inputs, while the other only with smaller ones. The `cut` bug can be found only when using two long arguments—but the original KLEE tests were using a single long argument—and the `printf` bug can only be found with an argument of size one—but the original KLEE tests used a larger size. Good regression test suites invoke applications with representative arguments, both in number and size, which ZESTI successfully exploits.

VIII. Conclusion

We have presented ZESTI, a lightweight symbolic execution-based tool that automatically improves regression test suites with the ability to reason about all possible input values on paths executed by the test suite, as well as explore additional paths around sensitive instructions. ZESTI approaches testing from two different angles: first, ZESTI significantly broadens the number of bug checks performed by a regression suite and therefore the number of bugs found. Second, by using the regression suites as a starting point, ZESTI provides an effective solution for guiding the exploration of the symbolic search space. As a result of these features, we were able to successfully apply ZESTI to three popular software systems—`GNU Coreutils`, `readelf`, and `libdwarf`—where it found 52 previously unknown errors, including two in the `Coreutils` suite, which was previously checked thoroughly via symbolic execution.

We believe our technique can be effectively integrated with existing regression suites, and could help bridge the gap between standard regression testing and symbolic execution, by providing a lightweight, incremental way of combining the two techniques.

We are making our tool ZESTI available as open-source at http://srg.doc.ic.ac.uk/projects/zesti.

Acknowledgments

We would like to thank Alastair Donaldson, Petr Hosek, Paul Thomson and the anonymous reviewers for their valuable comments on the text, and Alex Wolf for discussions about the relationship between our work and test augmentation. We are grateful to Daniel Dunbar for his fantastic work on KLEE, including the seeding functionality that ZESTI extends. This research has been supported by EPSRC through a DTA studentship and the grant EP/J00636X/1.

References

[1] S. Anand, C. S. Păsăreanu, and W. Visser, "JPF-SE: a symbolic execution extension to Java PathFinder," in *TACAS'07*, Mar.-Apr. 2007.

[2] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *ISSTA'10*, Jul. 2010.

[3] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *ISSTA'11*, Jul. 2011.

[4] T. Ball and S. K. Rajamani, "Automatically validating temporal safety properties of interfaces," in *SPIN'01*, May 2001.

[5] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in *POPL'93*, Jan. 1993.

[6] P. Boonstoppel, C. Cadar, and D. Engler, "RWset: Attacking path explosion in constraint-based test generation," in *TACAS'08*, Mar.-Apr. 2008.

[7] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *EuroSys'11*, Apr. 2011.

[8] W. R. Bush, J. D. Pincus, and D. J. Sielaff, "A static analyzer for finding dynamic programming errors," *Software: Practice and Experience*, vol. 30, no. 7, pp. 775–802, 2000.

[9] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI'08*, Dec. 2008.

[10] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice—preliminary assessment," in *ICSE Impact'11*, May 2011.

[11] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "Mace: model-inference-assisted concolic exploration for protocol and vulnerability discovery," in *USENIX Security'11*, Aug. 2011.

[12] M. Das, S. Lerner, and M. Seigle, "Path-sensitive program verification in polynomial time," in *PLDI'02*, Jun. 2002.

[13] J. S. Foster, T. Terauchi, and A. Aiken, "Flow-sensitive type qualifiers," in *PLDI'02*, Jun. 2002.

[14] P. Godefroid, "Model Checking for Programming Languages using VeriSoft," in *POPL'97*, Jan. 1997.

[15] P. Godefroid, "Compositional dynamic test generation," in *POPL'07*, Jan. 2007.

[16] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI'05*, Jun. 2005.

[17] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS'08*, Feb. 2008.

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

[19] M. J. Harrold, C. Unwersity, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM TOSEM*, vol. 2, pp. 270–285, 1993.

[20] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *USENIX-Winter'92*, Jan. 1992.

[21] G. J. Holzmann, "The model checker SPIN," *Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[22] N. L. Johnson, A. W. Kemp, and S. Kotz, *Univariate discrete distributions*. Wiley-Blackwell, 2005.

[23] P. Joshi, K. Sen, and M. Shlimovich, "Predictive testing: amplifying the effectiveness of software testing," in *ESEC/FSE'07*, Sep. 2007.

[24] J. C. King, "Symbolic execution and program testing," *CACM*, vol. 19, no. 7, pp. 385–394, July 1976.

[25] E. Larson and T. Austin, "High coverage detection of input-related security faults," in *USENIX Security'03*, Aug. 2003.

[26] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO'04*, Mar. 2004.

[27] R. Majumdar and K. Sen, "Hybrid concolic testing," in *ICSE'07*, May 2007.

[28] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[29] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI'07*, Jun. 2007.

[30] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE TSE*, vol. 22, 1996.

[31] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *ASE'08*, Sep. 2008.

[32] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *ESEC/FSE'05*, Sep. 2005.

[33] D. Song, D. Brumley, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *ICISS'08*, Dec. 2008.

[34] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *ISSTA'02*, Jul. 2002.

[35] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux, "eXpress: guided path exploration for efficient regression test generation," in *ISSTA'11*, Jul. 2011.

[36] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, Tech. Rep., 2002.

[37] N. Tillmann and J. De Halleux, "Pex: white box test generation for .net," in *TAP'08*, Apr. 2008.

[38] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *NDSS'00*, Feb. 2000.

[39] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: techniques and tradeoffs," in *FSE'10*, Nov. 2010.

[40] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," in *OSDI'04*, Dec. 2004.

[41] S. Yoo, M. Harman, and S. Ur, "Measuring and improving latency to avoid test suite wear out," in *ICSTW'09*, Apr. 2009.

[42] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE TSE*, vol. 28, no. 2, pp. 183–200, 2002.