Experience Report: How is Dynamic Symbolic Execution Different from Manual Testing? – A Study on KLEE

Xiaoyin Wang[1], Lingming Zhang[2], Philip Tanofsky[1]
[1]Department of Computer Science, University of Texas at San Antonio, TX 78249, USA
[2]Department of Computer Science, University of Texas at Dallas, TX 75080, USA
{xiaoyin.wang, philip.tanofsky}@utsa.edu, lingming.zhang@utdallas.edu

ABSTRACT

Software testing has been the major approach to software quality assurance for decades, but it typically involves intensive manual efforts. To reduce manual efforts, researchers have proposed numerous approaches to automate test-case generation, which is one of the most time-consuming tasks in software testing. One most recent achievement in the area is Dynamic Symbolic Execution (DSE), and tools based on DSE, such as KLEE, have been reported to generate test suites achieving higher code coverage than manually developed test suites. However, besides the competitive code coverage, there have been few studies to compare DSE-based test suites with manually developed test suites more thoroughly on various metrics to understand the detailed differences between the two testing methodologies. In this paper, we revisit the experimental study on the KLEE tool and GNU CoreUtils programs, and compare KLEE-based test suites with manually developed test suites on various aspects. We further carried out a qualitative study to investigates the reasons behind the differences in statistical results. The results of our studies show that while KLEE-based test suites are able to generate test cases with higher code coverage, they are relatively less effective on covering hard-to-cover code and killing mutants. Furthermore, our qualitative study reveals that KLEE-based test suites have advantages in exploring error-handling code and exhausting options, but are less effective on generating valid string inputs and exploring meaningful program behaviors.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability

Keywords

Manual Testing, Dynamic Symbolic Execution, Empirical Study

1. INTRODUCTION

In modern software development, software testing has been widely used to validate software quality. Conceptually, software testing is simple – (1) creating test cases and (2) executing those test cases to expose possible faults. However, in practice, software testing is quite complicated and requires intensive manual efforts. Test generation is one of the most important and expensive tasks in software testing. Therefore, researchers have proposed various approaches for automated test generation, such as guided random testing [31, 9], model-based testing [11, 35], symbolic execution [36, 7], etc. All these approaches have a large number of technical variants and have been evaluated on various software projects for their effectiveness, mainly based on code coverage [40], mutation scores [22], or real bugs [7] found.

Despite the enthusiasm in investigating advanced test-generation techniques, automated test generation techniques are not widely applied in practice compared with automated test-execution techniques (e.g., JUnit). For example, many open-source Java projects still use manually developed test cases, but have JUnit support for automated test execution. One well-known limitation for using automatically generated test cases is the lack of test oracles [37].

Besides the test oracle problem, it is also not clear whether automatically generated test suites are comparable to manually developed test suites on their test sufficiency. It should be noted that existing evaluations [7, 31] on automated test-case-generation techniques show that both automatically generated test suites and manually developed test suites are imperfect (not reaching 100% code coverage or mutation score). However, few studies have been directed to understand the practical value of automatically generated test suites by comparing them with manually developed test suites. It is not known that whether automatically generated test suites are able to cover most of code covered by manually developed test suites, and reveal the most of bugs revealed by manually developed test suites. Due to the known weakness [4] of existing test-sufficiency metrics (e.g., code coverage, mutation scores), simple comparison of values on these metrics are often not enough for understanding the whole picture, and deeper investigation of the program behavior during the execution of test suites often provides important information also.

To step towards the answers of the above questions, in this paper, we present an empirical study to compare test suites generated by Dynamic Symbolic Execution (DSE) with manually developed test suites. Actually, researchers have proposed numerous techniques for automatic test-case generation. As a first step, we focus on DSE-based test cases in this paper for the following reasons. First of all, DSE is commonly viewed as one of the most promising techniques on automatic test-case generation and has attracted lots of research interests [7, 42], thus representing state-of-the-art technique in the area. Second, due to the effectiveness of DSE, there are mature off-the-shelf tools (e.g., KLEE [7], Pex [39]) based on the technique. Third, as a fully-automatic and systematic technique for test-case generation, the results of DSE suffer from little noises from human intervention or instability.

Specifically, we perform our empirical study[1] by revisiting the application of KLEE on GNU CoreUtils programs. The reasons are as follows. First, we believe that KLEE test suites and manually developed test suites of CoreUtils represent the high end of their types. On one hand, KLEE is designed and configured specifically for CoreUtils, and has been reported to generate test suites that achieve higher code coverage than manually developed test suites [7]. On the other hand, GNU CoreUtils programs have been developed for decades and are widely used all over the world, so their test suites (bundled with their source code) should also be of high quality. Thus we believe that both test suites represent the high end of their types.[2] Second, for DSE to achieve high test coverage, proper modeling of library APIs is typically required, and this has been done in KLEE for CoreUtils. According to a previous study [41] performed on Pex, lacking of library API models is one of the major reasons why code is not covered. Note that, due to the prevalence of library APIs, they may cause DSE-exploration to stop early and mask other potential limitations of DSE techniques. Therefore, using the KLEE test suites on CoreUtils helps to rule out the effect of lacking library API models. Third, the KLEE tool has been tested on the CoreUtils projects and proved to be stable enough so that we may suffer less noises from software bugs.

Our empirical study consists of a quantitative study and a qualitative study. In our quantitative study, we first compared KLEE-based test suites with manually developed test suites on their code coverage and mutation scores (we exclude the test oracle factor by only using outputs to check mutation scores). Then, we further compared KLEE-based test suites with manually developed test suites on their test sufficiency by controlling their sizes, i.e., ensuring that they have the similar number of tests. After that, we defined hard-to-kill mutants and hard-to-cover code and compared two types of test suites on these more difficult testing problems. Moreover, we studied the overlap and difference on covered code and killed mutants between the two types of test suites to check how much extra value KLEE-based test suites can provide. In our qualitative study, we inspected the code covered by only one type of test suites, and mutants killed by only one type of test suites, to understand the reasons behind quantitative results.

This paper makes the following major contributions:

- We present a large-scale quantitative study to compare KLEE-based test suites with manually developed test suites on various test-sufficiency metrics (i.e., code coverage and mutation scores) and considering various factors (i.e., controlled test suite size, hard-to-cover code, hard-to-kill mutants).
- We present a study on the overlap and difference of the code/-mutants covered/killed by the two types of test suites to evaluate how much extra value KLEE-based test suites may provide on test sufficiency.
- We present a qualitative study including the inspection of thousands of lines of code to explain the quantitative results, and to find the difference between the two types of test suites on specific code covered and mutants killed.

[1]All data used in our study is available at `http://xywang.100871.net/testemp.html`

[2]Note that, the version history of GNU CoreUtils programs shows intensive human efforts on their test suites, indicating that the test suites are mainly manually developed, although usage of some helper tools can not be ruled out.

The key findings of our study are as follows.

- The code coverage of KLEE-based test suites (80.5% on average) is higher than that of manually developed test suites (70.9% on average). Despite the higher coverage, the mutation scores of KLEE-based test suites (51.3% on average) are lower than those of manually developed test suites (54.4% on average). When reducing the KLEE-based test suites to the size of corresponding manually developed test suites, the code coverage does not drop much, but the mutation scores drop from 51.3% to 42.8%.
- Manually developed test suites perform better than KLEE-based test suites on covering hard-to-cover code and killing hard-to-kill mutants.
- KLEE-based test suites are able to provide extra value to manually developed test suites, on both code coverage (18.3 extra percentage points), and mutation scores (11.4 extra percentage points).
- KLEE-based test suites are more effective than manually developed test suites on covering error-handling code and exhausting all possible input options.
- KLEE-based test suites are less effective on exploring some meaningful paths and generating valid string or structural inputs to go through the input parser.

2. STUDY DESIGN

In this section, we first present the research questions in our empirical study. Then, we introduce the leveraged metrics, the used tools, the subjects, and the definitions of hard-to-kill mutants and hard-to-cover code.

2.1 Research Questions

In our empirical study, we investigate the following research questions:

- **RQ1:** Are KLEE-based test suites comparable with manually developed test suites on test sufficiency?
- **RQ2:** For hard testing problems such as covering hard-to-cover code and killing hard-to-kill mutants, how do KLEE-based test suites compare with manually developed test suites?
- **RQ3:** How much extra value can KLEE-based test suites provide to manually developed test suites?
- **RQ4:** What are the characteristics of the code/mutants covered/killed by KLEE-based test suites, but not by manually developed test suites?
- **RQ5:** What are the characteristics of the code/mutants covered/killed by manually developed test suites, but not by KLEE-based test suites?

The first research question aims to provide the basic understanding of how KLEE-based test suites compare with manually developed test suites, and whether the former test suites can replace the latter. Also, KLEE-based test suites are usually larger than manually developed test suites and thus require more test oracles (which may be mainly constructed manually). Thus it is important to see whether KLEE-based test suites achieve test sufficiency high enough with a small test suite available (so that manual test oracle generation is possible). In our experiments, we use the coverage-based test prioritization technique (with the additional strategy [33]) to reduce KLEE-based test suites to the size of their corresponding manually developed test suites. Then we compare the test sufficiency of the reduced KLEE-based test suites (referred to as *comparative sufficiency / code coverage / mutation scores*) with that of manually developed test suites.

The second research question aims to understand how KLEE-based test suites perform on harder testing tasks. When it is harder for developers to write test cases to cover certain code or to kill certain mutants, helping to cover the code or to kill the mutants brings in more value. We will introduce how we define hard-to-cover code and hard-to-kill mutants in Section 2.4.

The third research question aims to understand whether KLEE-based test cases can bring extra value to human-written test cases. If KLEE-based test suites cover much code not covered by manually developed test suites or kill many mutants not killed by manually developed test suites, it implies that KLEE-based test suites can serve as a good supplement of manually developed test suites.

The fourth and the fifth research questions aim to reveal the facts behind the statistical results. By investigating the code and mutants that are covered or killed by one type of test suites but not by the other, we will be able to understand the major achievements and shortcomings of the current DSE techniques and tools.

2.2 Tools for the Study

For the reasons mentioned in Section 1, we choose to use KLEE [7] to generate DSE-based test suites in our study. Specifically, we followed the configuration and setup process in the official KLEE tutorial for testing CoreUtils[3]. For each subject in our experiment, we executed KLEE for 20 minutes and collected all test cases generated. The reason is that our KLEE-based test suites achieve similar code coverage when averagely 90% of test cases are reduced (will be shown in Table 2 later). This implies that our KLEE-based test suites already contain a large amount of redundant test cases and more testing time may not result in much enhancement. However, considering the variety of software project, a fixed timeout may not be sufficient for studying all different kinds of test subjects, so we may leverage some other termination criteria such as saturation criteria in future larger scale studies.

In our empirical study, we measured test sufficiency of test suites with code coverage and mutation scores. Specifically, for code coverage, we used the standard statement coverage [2], while for mutation score, we also used the standard definition, which is the proportion of killed mutants in all generated mutants [3, 5]. For the collection of code coverage and mutation scores, we used widely used mature tools to reduce potential errors in implementations. Specifically, we used gcov [2] to collect statement coverage, and mutGen [3] to generate mutants for the subject programs. For each subject, we randomly selected 100 mutants. For the subjects on which mutGen can only generate less than 100 mutants, we used all the generated mutants. To calculate mutation scores, following existing studies [27], we viewed console output of the original program as test oracles, and compared them with the console outputs of the mutants. A mutant is considered killed if there are any difference between its console output and the console output of the original program.

Table 1: Subject Statistics

Subject	LOC	# Mut	#KLEE	#Man
base64	3989	100	1918	150
basename	4026	100	1235	30
chcon	4343	100	521	7
chgrp	4278	100	543	59
cksum	3983	100	4	7
comm	3997	100	408	7
cut	4195	100	4872	220
dd	4734	100	3453	40
dircolors	4093	100	778	12
dirname	3889	87	1201	23
du	5790	100	828	77
env	3937	87	3074	7
expand	3916	100	780	11
expr	9565	100	420	154
fold	3891	100	4881	13
groups	4002	53	2177	12
link	3829	100	628	7
logname	3902	40	633	7
mkdir	4213	100	581	446
mkfifo	3959	84	674	17
mknod	3840	100	813	14
nice	4010	100	1441	50
nl	10037	100	6300	14
od	4463	100	582	20
paste	3837	100	1075	18
pathchk	3857	100	1140	13
printf	4251	100	23135	28
readlink	4154	89	8229	181
rmdir	3892	100	755	23
sleep	4199	100	640	23
split	4428	100	2470	22
sum	4068	100	3027	29
sync	3919	31	231	7
tee	3966	100	3051	20
touch	4744	100	7660	1252
tr	4150	100	713	1044
tsort	3856	100	706	17
unexpand	3903	100	8731	46
unlink	3865	71	608	7
wc	4075	100	17987	52

2.3 Subjects

In our empirical study, we used 40 CoreUtils (Version 6.11) programs as subjects. The basic information of the 40 programs is shown in Table 1. In Column 1-5, we present the program name, the program size (in Lines of Code, and including library code following previous work on KLEE [7, 27]), number of mutants used, the size of the KLEE-based test suite, and the size of the manually developed test suite, respectively.

Selection of Subjects. Note that we did not use all available CoreUtils programs as subjects. We did not include the rest of CoreUtils programs in our study, because those programs do not produce outputs or their outputs are related to environment (such as system time), making it hard to determine whether the mutation faults are revealed or not.

Size of Test Suites. For CoreUtils programs, most manually developed test cases are scripts involving a number of subject programs, so there can be different ways to count test cases. In our experiments, we count all distinct command line invocations of a certain subject program as different test cases. For example, the script `ls | wc` contains one test case for subject `ls`, and one test case for `wc`. Moreover, when counting the number of manually developed test cases, we only counted the test cases executed in our study, and did not count the test cases not executed because those test cases were designed for a different platform or require special configurations.

2.4 Hard-to-Cover Code and Hard-to-Kill Mutants

Code coverage and mutation scores actually measure all statements and mutants as equivalent with each other on their value. However, certain code/mutant can be much harder to be covered/killed than others. If automatic test-case generation tools are able to cover hard-to-cover code and hard-to-kill mutants, it may bring more value by saving developers more time on trying to cover/kill

[3]`http://klee.github.io/tutorials/testing-coreutils/`

those code and mutants. By contrast, if manually developed test suites cover/kill more hard-to-cover code/hard-to-kill mutants, it can show the directions in which KLEE-based techniques can be further improved.

To measure the difficulty of covering certain code, we leverage the idea as follows. If covering a statement requires the control flow to take certain branch outcome at more control points (i.e. conditional predicates), it is likely that more restricted inputs are required to cover the statement, and the statement is harder to cover. Therefore, we measure the difficulty to cover a statement with its depth in an Interprocedural Control Dependence Graph (ICDG) [10]. Typically, an ICDG also includes nodes that are not control point (e.g., method entries). Since only control points affect the difficulty of covering code under their control, we only count control points when calculating the depth of a statement. When there are multiple paths to reach a statement's corresponding node in the ICDG (from the main entry node), we always choose the path that contains the least number of control points to calculate the depth. In the rest of the paper, for brevity, we refer to the depth of a statement in ICDG (considering only control points) as the statement's ICDG-Depth. It should be noted that, it is very difficult to measure the difficulty for covering certain code. We believe that ICDG-Depth reflects code-covering difficulty to some extent, but there are also other factors (e.g., complexity of code and path constraints) that we may consider in future studies.

Figure 1 shows a sample ICDG generated by CodeSurfer [1] from the code portion below, and we highlight the control points with bold font. In the code portion, consider the statements at Line 6 and Line 14, whose corresponding nodes in ICDG have been marked with their line numbers. From the ICDG, we can see that the paths from the main entry node to the node marked with "Line 6" includes at least 1 control point (marked as red dashed line), while the paths from the main entry node to the node marked with "Line 14" includes at least 2 control points (marked as green dashed line). Therefore, the ICDG-Depths of the statement at Line 6 and Line 14 are 1 and 2, respectively.

```
1:  void main() {
2:        int sum, i;
3:    sum = 0;
4:    i = 1;
5:    while ( i<11 ) {
6:      sum = add(sum, i);
7:      i = add(i, 1);
8:    }
9:  }
10: int add(int a, int b){
11:   if (b > 0)
12:     return a + b;
13:   else
14:     return a;
15: }
```

For the difficulty of killing mutants, there are two aspects to be considered. The first aspect is whether the mutant is difficult to be covered, and the second aspect is whether the mutant is difficult to be killed once it is covered. Since the first aspect actually has been considered in hard-to-cover code, we consider only the second aspect in the definition of hard-to-kill mutants. Specifically, we define hard-to-kill mutants as mutants that are covered but not killed by both test suites.

3. QUANTITATIVE ANALYSIS

In this section, we present the results of our empirical study and how these results may answer the research questions (RQ1-RQ3) in Section 2.1.

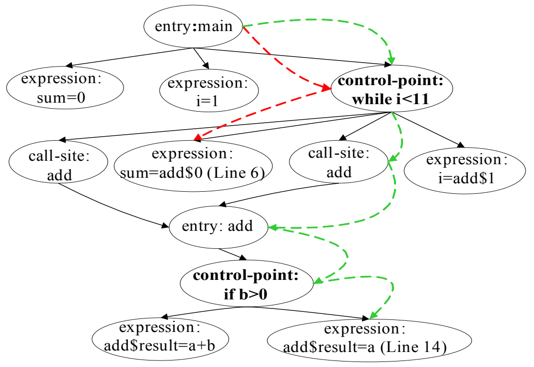

Figure 1: A Sample Inter-Procedure Control Dependence Graph

3.1 Test Sufficiency Comparison

In this subsection, we present the test sufficiency of KLEE-based test suites and manually developed test suites in Table 2, to answer RQ1. In Table 2, the first column presents the name of each subject. Columns 2-3 present the code coverage of KLEE-based test suites and manually developed test suites, respectively. The numbers in brackets in Column 2 are the comparative code coverage of KLEE-based test suites (code coverage when reduced to the size of corresponding manually developed test suites). Similarly, Columns 4-5 present the mutation scores of KLEE-based test suite, and manually developed test suites, respectively. The numbers in brackets in Column 4 are the comparative mutation scores of KLEE-based test suites (mutation scores when reduced to the size of corresponding manually developed test suites). As a reference, in Column 6, we present the ratio of the number of manual test cases to the number of KLEE-based test cases for each subject. From Table 2, we have the following four observations.

First of all, we confirm that the code coverage of KLEE-based test suites are better than that of manually developed test suites. Specifically, the average code coverage is 80.5% for KLEE-based test suites while it is 70.9% for manually developed test suites. Furthermore, KLEE-based test suites achieve higher code coverage in 28 of 40 subjects, same code coverage in 2 subjects, and lower code coverage in the remaining 10 subjects.

Second, we find that although KLEE-based test suites achieve higher code coverage, their average mutation score (51.3%) is lower than that of manually developed test suites (54.5%). Furthermore, manually developed test suites achieve higher mutation scores in 22 of 40 subjects, same mutation scores in 4 subjects, and lower mutation scores in the remaining 14 subjects.

Third, as expected, sizes of KLEE-based test suites are much larger than sizes of manually developed test suites. Column 8 shows that, on average, the sizes of manually developed test suites are only 10.2% of the size of KLEE-based test suites. In two subjects, the remaining percentages are 100%, indicating that the manually developed test suites are larger than KLEE-based test suites, so we did not do prioritization and reduction at all.

Fourth, from numbers in brackets, we observe that, after reducing the sizes of KLEE-based test suites to averagely 10.2% of their original sizes, the reduced test suites are able to achieve very similar code coverage (80.3%) compared with the original test suites (80.5%), the mutation scores drop more significantly than code coverage (i.e., from 51.3% to 42.8%).

3.2 Study on Hard Testing Problems

To answer RQ2, we compared the test sufficiency of KLEE-based test suite and manually developed test suite on the hard-to-

Table 2: Comparison on Test Sufficiency between KLEE-Based and Manually Developed Test Suites (%)

Subject	Coverage (%)		Killed Mutants (%)		#M/#K
	KLEE	Man	KLEE	Man	(%)
base64	89.5 (89.5)	78.1	59.0 (57.0)	83.0	7.8
basename	100.0 (100.0)	97.4	72.0 (32.0)	94.0	2.4
chcon	31.8 (29.2)	15.4	16.0 (16.0)	16.0	1.3
chgrp	68.9 (68.9)	76.7	28.0 (26.0)	75.0	10.9
cksum	56.5 (56.5)	69.4	0.0 (0.0)	0.0	100.0
comm	68.4 (67.3)	64.3	55.0 (55.0)	66.0	1.7
cut	77.4 (77.4)	89.9	76.0 (66.0)	83.0	4.5
dd	47.8 (47.8)	69.0	11.0 (11.0)	10.0	1.2
dircolors	88.8 (88.8)	81.4	66.0 (62.0)	99.0	1.5
dirname	100.0 (100.0)	96.8	90.8 (50.6)	93.1	1.9
du	70.1 (70.1)	73.4	51.0 (51.0)	66.0	9.3
env	100.0 (100.0)	66.7	85.7 (76.2)	61.9	0.2
expand	84.8 (84.8)	76.2	45.0 (39.0)	45.0	1.4
expr	36.1 (36.1)	74.3	32.0 (21.0)	58.0	36.7
fold	86.7 (86.7)	79.6	55.0 (49.0)	51.0	0.3
groups	94.6 (94.6)	59.5	88.7 (88.7)	35.8	0.6
link	96.4 (96.4)	64.3	86.0 (60.0)	46.0	1.1
logname	92.0 (92.0)	52.0	72.5 (62.5)	25.0	1.1
mkdir	84.8 (84.8)	92.4	61.0 (45.0)	86.0	76.8
mkfifo	91.5 (91.5)	83.0	27.4 (27.4)	64.3	2.5
mknod	73.8 (73.8)	73.8	77.0 (45.0)	61.0	1.7
nice	94.9 (94.9)	44.1	61.0 (43.0)	68.0	3.5
nl	86.7 (82.0)	42.2	63.0 (50.0)	30.0	0.2
od	57.8 (57.3)	44.6	8.0 (8.0)	14.0	3.4
paste	90.4 (90.4)	56.1	60.0 (51.0)	43.0	1.7
pathchk	72.0 (72.0)	40.9	26.0 (26.0)	48.0	1.1
printf	81.3 (81.3)	67.7	59.0 (51.0)	63.0	0.1
readlink	98.0 (98.0)	80.0	76.1 (65.9)	62.5	2.2
rmdir	77.8 (77.8)	79.2	46.0 (46.0)	67.0	2.9
sleep	67.4 (67.4)	67.4	29.0 (29.0)	51.0	3.6
split	89.4 (89.4)	86.2	33.0 (33.0)	58.0	0.9
sum	94.7 (94.7)	78.9	77.0 (55.0)	75.0	1.0
sync	100.0 (100.0)	65.0	29.0 (25.8)	29.0	3.0
tee	87.0 (87.0)	73.9	80.0 (58.0)	98.0	0.7
touch	81.3 (81.3)	75.7	10.0 (10.0)	25.0	16.3
tr	49.0 (49.0)	83.3	26.0 (24.0)	13.0	100.0
tsort	92.6 (92.6)	96.6	80.0 (77.0)	87.0	2.4
unexpand	88.7 (88.7)	82.0	76.0 (67.0)	58.0	0.5
unlink	100.0 (100.0)	72.0	18.3 (18.3)	53.5	1.2
wc	71.5 (71.5)	68.1	40.0 (34.0)	14.0	0.3
Avg.	**80.5 (80.3)**	**70.9**	**51.3 (42.8)**	**54.4**	**10.2**

cover code and hard-to-kill mutants. Figure 2 shows the comparison on hard-to-cover code. In the Figure, the green shadowed boxes show the code coverage of KLEE-based test suite, when different thresholds on the ICDG-Depth are used. The white boxes show the code coverage of manually developed test suites, when different thresholds on the ICDG-Depth are used. Each box plot shows the average (a small square in the box), median (a line in the box), and upper/lower quartile values for the corresponding code coverage.

From the figure, we observe that, for the statements with ICDG-Depth larger or equal to 4, manually developed test suites always achieves higher code coverage than KLEE-based test suites. Another observation is that, the code coverage of KLEE-based test suites drops as the ICDG-Depth of the code increases. The reason is that for dynamic symbolic execution, a higher ICDG-Depth means more complex constraints and harder solutions. By contrast, manually developed test suites achieve a similar or higher code coverage on code with a higher ICDG-Depth. This implies that, either a high ICDG-Depth does not cause much trouble to human testers, or human testers pay more attention to the hard-to-cover code. In sum, manually developed test suites perform better than KLEE-based test suites on the code with higher ICDG-Depth.

In Table 3, we present the comparison of KLEE-based test suites and manually developed test suites on their ability to kill hard-to-kill mutants. As we mentioned in Section 2.4, we define hard-to-kill mutants of a test suite as the mutants that are covered but not killed by both test suites. In our study, we focus on only the hard-to-kill mutants that are covered by both test suites. The reason is

Table 3: Comparison on Hard-to-Kill Mutants

Subject	Hard-to-kill Mutants Covered by Both(%)		
	K-M	M-K	Neither
base64	0.0	24.0	9.0
basename	0.0	22.0	4.0
chcon	0.0	0.0	0.0
chgrp	0.0	18.0	5.0
cksum	0.0	0.0	0.0
comm	5.0	13.0	2.0
cut	9.0	10.0	0.0
dd	4.0	2.0	0.0
dircolors	0.0	28.0	0.0
dirname	0.0	2.3	3.4
du	17.0	4.0	0.0
env	16.7	0.0	3.6
expand	0.0	0.0	0.0
expr	0.0	0.0	0.0
fold	8.0	8.0	1.0
groups	32.1	0.0	0.0
link	11.0	8.0	2.0
logname	0.0	2.5	2.5
mkdir	4.0	19.0	1.0
mkfifo	0.0	33.3	9.5
mknod	18.0	4.0	11.0
nice	1.0	8.0	2.0
nl	21.0	3.0	0.0
od	0.0	6.0	0.0
paste	8.0	0.0	19.0
pathchk	0.0	22.0	8.0
printf	7.0	9.0	0.0
readlink	13.6	2.3	0.0
rmdir	3.0	4.0	1.0
sleep	2.0	13.0	5.0
split	2.0	27.0	20.0
sum	7.0	8.0	1.0
sync	0.0	0.0	3.2
tee	1.0	9.0	0.0
touch	0.0	14.0	30.0
tr	18.0	3.0	19.0
tsort	5.0	11.0	7.0
unexpand	17.0	0.0	0.0
unlink	0.0	35.2	16.9
wc	14.0	1.0	18.0
Avg.	**6.1**	**9.3**	**5.1**

that a hard-to-kill mutant that is covered by only one test suite will be killed by at most one test suites, and thus it is hard to compare test suites on it. The hard-to-kill mutants that are covered by both test suites fall into three categories, the mutants killed by the KLEE-based test suites but not by manually developed test suites (K-M), the mutants killed by manually developed test suites but not by KLEE-based test suites (M-K), and the mutants that are killed by neither test suite (Neither). In Columns 2-4 of Table 3, we present the portion of mutants in the K-M category, the M-K category, and the Neither category, respectively. The results show that, on average, the proportion the M-K category is larger (by 3.2 percentage points) than that of K-M category. Among all 40 subjects, M-K category is larger in 22 subjects, while K-M category is larger in 12 subjects. These observations indicate that, when both test suites are able to cover a mutant, manually developed test suites have a better chance to kill it.

3.3 Overlap Study

To answer RQ3, we measure to what extent the two types of test suite overlap with each other and complement each other on test sufficiency. Specifically, we can measure the overlap portion of the two types of test suites on covered code and killed mutants, as well as the portion of covered code and killed mutants by only one type of test suite. In Columns 2-4 of Table 4, we provide the portion of code covered by both types of test suites, the portion of code covered by only KLEE-based test suites, and the portion of code covered by only manually developed test suites. Similarly, in Columns 5-7 of Table 4, we provide the portion of mutants killed

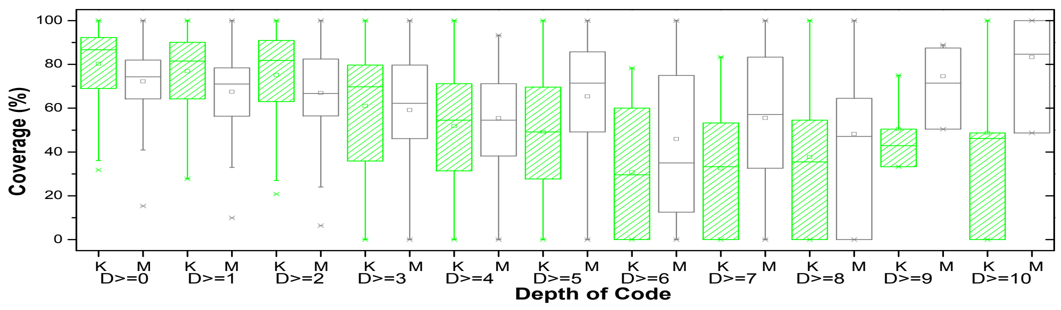

Figure 2: The comparison of coverage achieved by KLEE and Manual test suites on code of different ICDG-Depths

Table 4: Overlap and Difference of Covered Code and Killed Mutants

Subject	Coverage			Killed Mutants		
	K∩M	K-M	M-K	K∩M	K-M	M-K
base64	78.1	11.4	0.0	59.0	0.0	24.0
basename	97.4	2.6	0.0	72.0	0.0	22.0
chcon	13.8	17.9	1.5	16.0	0.0	0.0
chgrp	52.2	16.7	24.4	26.0	2.0	49.0
cksum	56.5	0.0	12.9	0.0	0.0	0.0
comm	46.9	21.4	17.3	48.0	7.0	18.0
cut	75.0	2.4	14.9	66.0	10.0	17.0
dd	46.5	1.2	22.5	7.0	4.0	3.0
dircolors	76.6	12.2	4.8	66.0	0.0	33.0
dirname	96.8	3.2	0.0	90.8	0.0	2.3
du	61.5	8.6	12.0	34.0	17.0	32.0
env	66.7	33.3	0.0	61.9	23.8	0.0
expand	70.9	13.9	5.3	45.0	0.0	0.0
expr	33.7	2.4	40.5	32.0	0.0	26.0
fold	69.0	17.7	10.6	38.0	17.0	13.0
groups	59.5	35.1	0.0	35.8	52.8	0.0
link	60.7	35.7	3.6	38.0	48.0	8.0
logname	52.0	40.0	0.0	22.5	50.0	2.5
mkdir	77.3	7.6	15.2	57.0	4.0	29.0
mkfifo	76.6	14.9	6.4	27.4	0.0	36.9
mknod	56.3	17.5	17.5	57.0	20.0	4.0
nice	44.1	50.8	0.0	56.0	5.0	12.0
nl	36.0	50.7	6.2	21.0	42.0	9.0
od	38.2	19.6	6.3	8.0	0.0	6.0
paste	56.1	34.2	0.0	41.0	19.0	2.0
pathchk	40.9	31.1	0.0	26.0	0.0	22.0
printf	58.4	23.0	9.3	42.0	17.0	21.0
readlink	78.0	20.0	2.0	59.1	17.0	3.4
rmdir	62.5	15.3	16.7	42.0	4.0	25.0
sleep	52.2	15.2	15.2	27.0	2.0	24.0
split	82.0	7.4	4.1	31.0	2.0	27.0
sum	78.9	15.8	0.0	65.0	12.0	10.0
sync	65.0	35.0	0.0	29.0	0.0	0.0
tee	73.9	13.0	0.0	79.0	1.0	19.0
touch	67.4	13.9	8.3	10.0	0.0	15.0
tr	44.7	4.3	38.5	8.0	18.0	5.0
tsort	90.6	2.0	5.9	75.0	5.0	12.0
unexpand	76.8	11.9	5.2	54.0	22.0	4.0
unlink	72.0	28.0	0.0	18.3	0.0	35.2
wc	47.7	23.8	20.4	7.0	33.0	7.0
Avg.	**62.2**	**18.3**	**8.7**	**39.9**	**11.4**	**14.5**

by both types of test suites, and one type of test suite only. For brevity, in the title lines of the table we use "K" as the abbreviation of "KLEE-based", and "M" as the abbreviation of "Manual". Then, for example, K∩M indicates all code or mutants that are covered or killed by both KLEE-based test suites and manual test suites. From Table 4, we can observe that, KLEE-based test suites provide extra value to manually developed test suites. Specifically, KLEE-based test suites cover 18.3% extra code and kill 11.4% extra mutants on average. Furthermore, KLEE-based test suites cover extra code in all 40 subjects, and kill extra mutants in 25 of 40 subjects.

3.4 Summary

In this subsection, we summarize the findings of our quantitative study and the answers of research questions RQ1 through RQ3.

Comparison on Test Sufficiency. Our experiments show that KLEE-based test suites are able to achieve higher coverage than manually developed test suites, but their mutation scores are lower. This observation implies that code coverage is not sufficient in evaluating test suites. KLEE-based techniques target at high code coverage and that is achieved, but the generated test suites may be not as effective on revealing software bugs. It is possible to have KLEE-based techniques to target at pre-planted mutants [43], but we may still not guarantee that the generated test suites can kill other mutants or real bugs. Reducing KLEE-based test suites to the size of manually developed test suites cause a drop of 9 percentage points on mutation scores. This implies that, to achieve similar test sufficiency, human testers may spend more time on constructing test oracles for KLEE-based test suites if automatic test oracles are not available.

Hard-to-Cover Code and Hard-to-Kill Mutants. Our experiments show that manually developed test suites perform better on both hard-to-cover code and hard-to-kill mutants. More advanced techniques are still required to enhance KLEE-based test suites for hard testing problems, or it may be wise to have human testers help on hard testing problems [41].

Replacement or Complement. Our experiments show that KLEE-based test suites and manually developed test suites tend to cover the same code and kill the same mutants. However, KLEE-based test suites can still provide extra values on both code coverage and mutation scores.

3.5 Threats to Validity

3.5.1 *Construct Validity*

Threats to construct validity are concerned with whether the experimental setup and metrics reflect real-world situations. The major threat to construct validity for our study is the metrics that we used for test sufficiency. To reduce this threat, following existing studies [18, 30, 23], we used the widely used statement coverage and mutation testing for assessing test sufficiency. However, although mutation testing has been shown to be suitable for software-testing experimentation [3, 12, 5], mutation faults are not exactly the same with real faults. Future reduction of this threat requires additional studies on subject systems with real bug information.

3.5.2 *Internal Validity*

Threats to internal validity are concerned with the uncontrolled factors that may be also responsible for the results. The major threat to internal validity for our study is the potential faults in our implementations and the tools that we used to carry out our experiments. To reduce this threat, we used mature tools that have been widely used in software engineering research in our experiments, e.g., mutGen [3], gcov [2], and KLEE [7]. We also did our best to check and remove all errors from our code for statistics and comparison.

3.5.3 External Validity

Threats to external validity are concerned with whether the findings in our experiments are generalizable for other settings. Our subject programs, their original test cases, and the used test generation tool may all pose threats to external validity. First, although we used 40 Coreutils programs of various sizes, the differences seen in our study may be difficult to generalize to other C programs. Furthermore, our results may not generalize to programs written in languages other than C. Second, the results may not be generalizable to other test cases or other test generation tools. Further reduction of these threats requires additional studies involving additional programs, tests, and test-generation tools.

4. QUALITATIVE STUDY

The quantitative study presented in the previous section provides statistical results on how KLEE-based test suites compare with manually developed test suites. In this section, to investigate the underlying reasons of the statistical results (answering RQ4 and RQ5), we present a qualitative study on the covered code and killed mutants of the two types of test suites. For brevity, we omit some detailed information of the studied code chunks and mutants[4], and present only the major findings.

4.1 Code

Our quantitative study shows that, on code coverage, KLEE-based test suites achieve higher code coverage, but manually developed test suites perform better on the code that has higher ICDG-Depth. To explain this, we investigate selected samples of the KLEE-Man code chunks (code chunks covered by KLEE-based test suites but not by manually developed test suites) as well as Man-KLEE code chunks (code chunks covered by manually developed test suites but not by KLEE-based test suites).

4.1.1 KLEE-Man Code Chunks

When inspecting KLEE-Man code chunks, our first purpose is to find out what kind of code is largely missed by manually developed test suites, but covered by KLEE-based test suites. So, first of all, we inspected five subjects that have highest proportion of KLEE-Man covered code (See Table 4): `nl`, `nice`, `logname`, `link`, and `group`. The inspection results are as follows.

In subject `nl`, the major reason of much KLEE-Man code is that, the manually developed test suite fails to exercise 8 of the 11 command-line options, and thus is not able to cover all the relevant code. In subject `nice` (`nice` supports scheduling adjustment of processes), the manually developed test suite fails to provide any processes requiring scheduling adjustments. In the other three subjects, the test suites are not able to cover a series of common error scenarios such as invalid options, extra or less operands, input / output errors, etc.

From this inspection, we observe that, the major cause of KLEE-Man covered code is that manually developed test suites fail to cover some features or error scenarios. Thus, it is highly probable that software testers are aware of such code chunks, but they did not perform corresponding testing to cover them for some reasons which may include lack of time, confidence with some simple error-handling code, or belief in the rare usage of features / options. Therefore, when it is tedious for testers to write test cases to cover certain features or large number of options, KLEE (or other DSE-based tools) may be a good choice to save testing efforts. Also, it seems that the major reason why KLEE-based test suites achieve a much higher code coverage than manually developed test suites is that developers intentionally did not test some code.

[4]Detailed qualitative study results are available at `http://xywang.100871.net/testemp.html`.

Table 5: Inspection of KLEE-Man Code Chunks

Subject	Reason
dd	Fail to generate an invalid parameter that cannot be converted to long
expand	Fail to provide invalid characters in the “tabstops” parameter
expr	Fail to provide input with syntax errors on bracket matching
printf	Fail to explore the printing of most escape characters
paste	Fail to generate a file read error

Although manually developed test suites outperform KLEE-based test suites on code with high ICDG-Depth, we still want to see whether KLEE-based test suites are able to provide some additional value in covering the corner-cases in the hard-to-cover code. So we inspect the 5 longest[5] KLEE-Man code chunks which have high ICDG-Depth (> 4), and present the results in Table 5. From the table, we can see that 4 of the 5 code chunks (except the 4th) are error handling code, and human testers did not provide proper invalid inputs to explore them. Actually, it may be more difficult for human testers to come up with invalid test inputs, especially if the testers are also developers, because their concern is more on how to make the program work and it is also hard for them to consider invalid inputs since they are too familiar with the valid input requirements. Therefore, they wrote error-handling code to handle unexpected scenarios, but they may not know how to cover such code (or even whether such code is reachable).

In sum, KLEE-based test suites may be a good choice for covering large number of options or error scenarios, they are also able to provide some extra value on the hard-to-cover code, especially on the error-handling code.

4.1.2 Man-KLEE Code Chunks

Manually developed test suites perform better on hard-to-cover code, so we inspected 10 longest Man-KLEE code chunks (in different subjects) which have high ICDG-Depth (> 4), and the inspection results are presented in Table 6. From the table, we observe that, the reason why Man-KLEE code chunks are not covered is mostly that a string-type input with certain format is required, and KLEE-based test suites are not able to provide such proper string to pass certain type of tokenization or parsing. One of the code chunk (i.e. in `tsort`) is not covered because KLEE-based test suites fail to generate a proper tree structure. Since in `tsort`, the tree structure is parsed from a string, the root cause of missing this code chunk is still failing to provide a proper string. More recent researchers on DSE have developed a number of novel techniques [42, 24, 6] to present and solve string constraints. However, we are not able to directly test these techniques on CoreUtils subjects due to program language differences (none of the above techniques are for C), and difficulties in modeling relevant library functions. We will qualitatively discuss the potential effectiveness of these techniques on CoreUtils programs in Section 5.

In sum, code chunks are covered by manually developed test suites but not by KLEE-based test suites mostly because the exploration of these code chunks requires an input with relatively complex structures, such as formatted strings with specific structure.

[5]To cover more subjects, if a subject has more than 1 code chunk among the 5 longest, we only use the longer one, and inspected one more longest code chunk instead.

Table 6: Inspection of Man-KLEE Code Chunks

Subject	Reason
cut	Fail to provide an input string can be successfully tokenized with delimiters
dd	Fail to provide an input string parsed successfully with the symbol table
dircolors	Fail to have special characters in proper position in the input
expand	Fail to provide an option parameter parsed successfully to multiple tab stops
expr	Fail to provide an input string with ":" parsed successfully
fold	Fail to have ' ' in the required position in the input
wc	Fail to provide an input string containing proper delimiters
rmdir	Fail to provide a valid full path
tr	Fail to have special characters in proper place in the input
tsort	Fail to generate a tree structure requiring double rotation in balancing

4.2 Mutants

Our quantitative study shows that, although achieving higher code coverage, KLEE-based test suites have relatively lower mutation scores. In this study, we inspect the KLEE-Man mutants and Man-KLEE mutants to find out the reason behind this phenomena. We focus on only mutants that are covered by both types of test suites, but are killed by only one of them. The reason is that, the mutants not covered by a test suite will never be killed by it, so reasons for killing / failing to kill such mutants are not interesting and are more of code coverage issues, which have been addressed in the previous subsection.

4.2.1 KLEE-Man Mutants

To investigate the mutants that are killed by KLEE-based test suites but not by manually developed test suites, we manually inspected 10 randomly-selected KLEE-Man mutants, and present the inspection results in Table 7. From Table 7, we can see that there are three reasons causing the inspected KLEE-Man mutants. The first reason is that, the mutation is changing the error condition of a piece of error-handling code that has never been covered by manually developed test suites. For example, in the following code sample, a mutant that changes `optind + 1 < argc` to `optind + 2 < argc` can only be killed when `optind + 2 == argc`.

```
1: if(optind + 1 < argc){
2:     error (0, 0, _("extra␣operand␣\%s"), quote (argv
    [optind + 1]));
3: }
```

However, the error in Line 2 has never been covered by the manually developed test suite, and `optind + 2 == argc` never holds. Therefore, the manually developed test suite is not able to kill such mutants. The second reason is that, the mutation is deleting an assertion that is never violated during the execution of manually developed test suite. The third reason is that, the mutation causes the value of a variable to change. However, the variable is used in only some code that are never covered by the manually developed test suites.

In sum, the commonality of the three reasons is that, the mutation is affecting code / branches that are not covered by the manually developed test suite, so the root cause is the relatively low code coverage of manually developed test suites.

Table 7: Inspection of KLEE-Man Mutants

Subject	Reason
groups	change error conditions of errors never covered
cut	delete an assertion never violated
env	change a variable whose affected code never covered
fold	change error conditions of errors never covered
link	change error conditions of errors never covered
unexpand	change error conditions of errors never covered
sum	change error conditions of errors never covered
nl	change a variable value whose affected code never covered
paste	change error conditions of errors never covered
mkdir	change a variable value whose affect code never covered

4.2.2 Man-KLEE Mutants

The most interesting part of the qualitative study is the inspection of Man-KLEE mutants. In the quantitative study, we show that manually developed test suites achieve a higher mutation score despite low code coverage. This fact indicates that quite some mutants are covered but not killed by the KLEE-based test suites. We inspected 10 randomly selected Man-KLEE mutants to find out some potential reasons. The inspection results are presented in Table 8.

From Table 8, we can see that there are five different reasons for the 10 Man-KLEE mutants.

The first reason is that the mutation affects only code that are never covered by the KLEE-based test suites, which is similar to the cases that we introduced in the previous subsection. This reason accounts for 3 of the 10 mutants. The second reason is that, the mutants can be killed only when a specific path of the program is executed. This reason also accounts for 3 of 10 mutants, and we will present an example in the next paragraph to further illustrate this case. The third reason is that, a mutation cause the program to exit with error status. So the mutant can be killed only when a test case covers the mutation and exits with success status. However, all the KLEE-based test cases covering the mutation originally exit with error status (for other reasons). Thus the mutant cannot be killed, and we refer this scenario as *error overlapping*. The fourth reason is that, a trivial value (i.e. 0) is always provided as the input to a function. Since the product of 0 and any number is still 0, the mutated value of other variables is masked by multiplying with 0. The fifth reason is that, the KLEE-based test cases always provide a list with length 0 or 1 to an iteration loop, so that the loop is always executed 0 times or once. Therefore, when the loop condition is mutated to run the loop for only once (i.e., remove the while condition), the KLEE-based test suite cannot kill the mutant. It should be noted that, both the third and the fifth reason are also related to failing to execute a specific path.

We present a code sample (from basename.c) below to illustrate a meaningful path that is not explored by the KLEE-based test suite. The code sample is a function that removes a suffix (parameter `suffix`) from a name (parameter `name`). The function works only when the parameter `name` ends with the parameter `suffix`. If so, the while loop will be executed and a pointer will go from the end of `name` to the position where the suffix can be truncated. After that, `name` is truncated (Line 11). If parameter `suffix` is an empty string, the while loop will be skipped and only Line 10 and 11 are executed, but `name` is not truncated because the pointer is not moved. If `suffix` is neither empty nor a true suffix of `name`,

Table 8: Inspection of Man-KLEE Mutants

Subject	Reason
base64	the affected branch is not covered
basename	a specific path is never executed
chgrp	a specific path is never executed
dircolor	a specific path is never executed
link	error overlapping
rmdir	the affected branch is not covered
split	the affected branch is not covered
sum	trivial value
touch	error overlapping
tsort	a loop is always executed 0 or 1 time

the function will return (Line 8) after 1 or more iterations and nothing is changed. We can see that, the path <(6, 7)*, 10, 11> is the most meaningful path that performs the major design purpose of the function. However, the KLEE-based test suite is not able to cover the path although it covers all the statements and branches of the code sample by providing an empty suffix (covering <10, 11>), and invalid suffixes (covering <(6, 7)*, 8>).

```
1: void remove_suffix (char *name, const char *suffix)
    {
2:   char *np;
3:   const char *sp;
//put pointer np at the end of name
4:   np = name + strlen (name);
//put pointer sp at the end of suffix
5:   sp = suffix + strlen (suffix);
/*move pointers back iteratively to check whether name
    ends with suffix, do not do anything and return if
     name does not end with suffix*/
6:   while (np > name && sp > suffix){
7:     if (*--np != *--sp)
8:       return;
9:   }
/*if name is longer than suffix, truncate name at the
    current pointer*/
10:  if (np > name)
11:    *np = '$\slash$0';
12:}
```

In sum, the majority (60%) of the Man-KLEE mutants are caused by the failure of KLEE-based test suite to explore a specific program path (typically a meaningful path related to the second, third and fifth reasons).

4.3 Limitations

In our qualitative study, we inspected only a selected sample in each category of code chunks or mutants. Therefore, it is possible that our findings are not precise for the whole data set. We try to make the selected samples more representative by using random selection for mutants and selecting longest code chunks. It should be noted that, given the large size of test suites and code base, it is very difficult and time-consuming to manually inspect why certain code or mutant is not covered or killed by a whole test suite. During the study, we did much on-demand instrumentation (recording and checking values of relevant variables) to help the inspection, and spent large amount of time on investigating the code and logs.

5. LESSONS LEARNED

Practical Value of Current DSE-Based Test Suites. Our study confirms that, KLEE-based test suites achieve a competitive code coverage, compared with manually developed test suites. In particular, KLEE-based test suites have advantages on exploring large number of options, and providing various invalid inputs to cover error-handling code that manually developed test suites fail to cover. We guess that developers may be not good at providing invalid inputs because it requires them to think from a reverse side. Another possible case is that, when there are many options and much error-handling code that are not very complex, the developers feel it too tedious to test all cases. Anyway, in practice, KLEE-based test suites can definitely help developers to cover multiple options and error-handling code in testing, and revealing corresponding bugs. Also, various invalid inputs help to reveal security bugs (vulnerabilities) which are often invoked by well-designed invalid inputs. Thus, another practical usage of KLEE-based test suites may be detection of security bugs.

However, despite the high code coverage and acceptable mutation score achieved, the results of our study show that KLEE-based test suites fail to generate various valid inputs for many subjects studied. Some major difficulties include the inability of KLEE-based test suites to pass a complex input parser, or the inability of KLEE-based test suites to cover a meaningful path. Our discussion below shows that, some of these difficulties may be not specific to KLEE implementation, but may be a common weakness of other DSE techniques (i.e., constraint expressiveness and search strategies). It should be noted that testing with various valid inputs is typically more important than testing with invalid inputs, because most users , in most cases, will use software with valid inputs. Therefore, developers need to be careful, if they want to apply only DSE-based test suites to functional testing.

Handling String Inputs with Complex Formats. Our study shows that, one major reason why KLEE cannot cover all the code is that, the program to be tested accepts an input with complex format, and parses the input to determine what task to perform. In such a case, the program functions normally only if the given input passes the parsing process, and certain behaviors can be covered only when the input represents certain semantics. Since KLEE handles string inputs as character arrays, and array indexes are often simply increased when processing the array, a specific fragment of the input can be expressed only as a sub-character array of fixed index and size, and thus may cause some constraints for branch-flipping to be unsolvable. For example, in an input “a+b”, if a variable `op` representing the “+” operator is expressed as `input[1, 2]`, the constraint `op == '<<'` has no solution.

Recently, there have been a number of techniques to construct and solve string-related constraints, such as HAMPI [24], Kaluza [34], PASS [26] and the string resolver in Pex [6]. However, the string expressions supported by these research efforts are mainly designed for expressing the composition of several inputs, instead of a logical fragment of one long input. For example, it is easy to express `aaa:bbb` with two inputs `x = aaa`, and `y = bbb` as `x+:+y` (using “+” for concatenation), but it is very difficult to express the fragment `a+b` (the second argument of the invocation to function `f`) in one input `x=f(a, a+b)-g(c);`. Actually, none of the techniques mentioned above support common string operations for parsing, such as splitting and tokenization. If developers directly manipulate characters in string values for parsing, the construction and solution of the string-related constraints can be even more difficult.

As far as we know, there have been no effective constructors and solvers for string constraints generated in the input parsing process, which is mainly about logically expressing a fragment in a long structured input.

Formal specifications of the input (e.g., syntax trees) will definitely help, and there are some techniques [19, 28] taking advantage of them, but such specifications may be hard to be extracted automatically. Also, it should be noted that, since most CoreUtils programs use formatted strings as their input, the input parsing problem may be over-presented in our study, but any program that parses string input may face the problem mentioned above.

Detecting Meaningful Paths. Our study reveals that KLEE-based test suites fail to explore some meaningful paths in the program. As shown in the code sample in Section 4.2.2, it is possible to cover all statements and branches without exploring the major feature of a program, because only some specific paths in the program are meaningful. Since typical DSE searching strategies try to cover the code or branch never covered or taken, it may not be able to explore the meaningful paths. It should be noted that, missing meaning paths is very dangerous, because there are no alarms showing up in the code coverage, and users of DSE-based test suites may mistakenly believe that all behaviors of the program are explored.

Due to the path explosion problem, it is impossible to cover all the paths. Therefore, a more advanced searching strategy that prioritizes program paths according to their meaningfulness may be desired. It is not easy to identify meaningful paths, because the definition of meaningfulness differs in different programs. Some of intuitive characteristics of meaningful paths include: larger path length, more data dependencies along the paths, not ending with error-handling code, etc.

Another idea is to use some manually developed test cases as the seed test cases to boost DSE. In this way, manually developed test cases have higher probability to cover meaningful paths of the program, and then the DSE technique can help to further explore the corner cases.

6. RELATED WORKS

We have been aware of several existing research efforts on comparing automatically generated test cases and manually developed test cases. Specifically, Kracht et al. [25] applied EvoSuite [14] on a number of software projects with manual test cases, and reported that the two types of test cases are similarly effective. Compared to that work, our study considers a different test-case-generation technique (i.e., DSE), and conducted more detailed investigation on different code depths, test-suite complementarity, hard-to-kill mutants, as well as the qualitative features of the test suites. Ceccato et al. [8] compared automatically generated test suites and manually developed test suites on their helpfulness in debugging. Another closely related work was done by Fraser et al. [16]. They empirically investigated the impacts of tool supports in writing tests, and found that, testing tools do not provide significant help to human testers. In their study, they compared manual test cases written with and without tool supports, the test cases were created by study participants rather than the original developers.

Automated test generation has been largely explored in both industry and academia. Recent decades have seen many advances in white-box test generation. The existing white-box test generation techniques can be mainly categorized as systematic techniques or unit-level techniques. For systematic test generation, symbolic execution is one of the most widely studied techniques. Traditional symbolic execution techniques encode program path conditions into constraints and then use off-the-shelf solvers to generate test data for each program path to achieve high code coverage (e.g., DART [20], CUTE [36], Pex [39], KLEE [7]). Unit-level test generation techniques usually generate unit tests in the form of method invocation sequences. Pacheco et al. [31] aims to build unit tests incrementally by randomly selecting a method call to apply and finding parameters from previously-constructed test cases. Fraser et al. [14] used evolutionary search approach to iteratively generate and optimize unit test suites towards a certain coverage criterion. Later, Fraser et al. [17] further use mutant killing to guide the search-based test generation.

Since many automated test generation techniques use coverage to guide generation, many studies have been conducted to investigate the correlation between coverage criteria and test effectiveness. Frankl and Lakounenko [13] empirically evaluated the fault-detecting ability of two white-box test criteria: branch coverage and all-uses data flow coverage, and showed that tests achieving high branch or all-uses data flow coverage also have a higher probability to detect real faults. The study by Gligoric et al. [18] also confirmed that branch coverage is highly correlated with test effectiveness. Namin et al. [30] empirically studied the relation between test size, coverage, and test effectiveness, and found that both size and coverage are important to test effectiveness in fault detection. However, a recent study by Inozemtseva and Holmes [23] showed that coverage is not strongly correlated with test effectiveness. This inconsistency may be due to the different characteristics of tests not captured by coverage, e.g., how the tests were constructed: manually or automatically. This further motivates our study.

There are also studies investigating the testing status of real-world open source projects. Singh et al. [38] empirically explored 20,000 open-source projects, and found that bigger projects have a higher probability to contain test cases. Greiler et al. [21] explored the current testing practices currently used for the specific plug-in systems. Mostafa and Wang [29] studied the usage of mocking frameworks in real world software test code. Pham et al. [32] investigated how social coding sites influence testing behavior. They found several strategies that software developers and managers can use to positively influence the testing behavior. Fraser et al. [15] empirically evaluated automated test generation on 100 real-world Java projects. They found that high coverage is achievable on commonly used types of classes, and also identified future directions to improve code coverage. In this paper, we aim to investigate the differences between manual tests and DSE-based tests for real-world systems.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we carried out a comparison study on KLEE-based test suite and manually developed test suites. Specifically, the major findings of our study include: KLEE-based test suites are able to achieve higher code coverage but relatively lower mutation scores; KLEE-based test suites are able to provide extra value on both code coverage and mutation scores; KLEE-based test suites are not as good as manually developed test suites on hard-to-cover code and hard-to-kill mutants; and KLEE-based test suites are more effective on testing error handling code and exploring options, but are less effective on testing some program functions that require a specific path to be explored or an input with certain structure.

In the future, we plan to extend our work in the following directions. First of all, we plan to perform a larger scale quantitative and qualitative study considering more factors (e.g., more test-termination criteria, more measurements of the code-covering difficulty) to find more accurate and general conclusion on how manual and KLEE-based test suites differ in covering code and killing mutants. Second, we plan to compare the test suites generated by other DSE-based techniques with manually developed test suites. Third, our study identifies future directions to improve the state-of-art DSE techniques, we plan to develop more sophisticated techniques to follow those directions.

Acknowledgment

The authors from University of Texas at San Antonio are supported in part by NSF grant CCF-1464425, and DHS grant DHS-14-ST-062-001.

8. REFERENCES

[1] Codesurfer, http://www.grammatech.com/research/technologies/codesurfer.

[2] Gnu gcov for gcc, http://gcc.gnu.org/onlinedocs/gcc/gcov.html.

[3] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? [software testing]. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402–411, May 2005.

[4] J. Andrews, L. Briand, Y. Labiche, and A. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, Aug 2006.

[5] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, 2006.

[6] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, pages 307–321, 2009.

[7] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, 2008.

[8] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella. An empirical study about the effectiveness of debugging when random test cases are used. In *Proceedings of the 34th International Conference on Software Engineering*, pages 452–462, 2012.

[9] T. Chen. Adaptive random testing. In *Quality Software, 2008. QSIC '08. The Eighth International Conference on*, pages 443–443, Aug 2008.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[11] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: A systematic review. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on automated Software Engineering (ASE) 2007*, WEASELTech '07, pages 31–36, 2007.

[12] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *Software Engineering, IEEE Transactions on*, 32(9):733–752, 2006.

[13] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 153–162, 1998.

[14] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[15] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 178–188, 2012.

[16] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 291–301, 2013.

[17] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *Software Engineering, IEEE Transactions on*, 38(2):278–292, 2012.

[18] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 302–313. ACM, 2013.

[19] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 206–215, 2008.

[20] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223, 2005.

[21] M. Greiler, A. van Deursen, and M. Storey. Test confessions: a study of testing practices for plug-in systems. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 244–254, 2012.

[22] W. Howden. Weak mutation testing and completeness of test sets. *Software Engineering, IEEE Transactions on*, SE-8(4):371–379, July 1982.

[23] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Software Engineering (ICSE), 2014 36th International Conference on*, page to appear, 2014.

[24] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *ISSTA*, pages 105–116, 2009.

[25] J. Kracht, J. Petrovic, and K. Walcott-Justice. Empirically evaluating the quality of automatically generated and manually written test suites. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 256–265, 2014.

[26] G. Li and I. Ghosh. Pass: String solving with parameterized array and interval automaton. In *Hardware and Software: Verification and Testing*, pages 15–31. 2013.

[27] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 19–32. ACM, 2013.

[28] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 134–143, 2007.

[29] S. Mostafa and X. Wang. An empirical study on the usage of mocking frameworks in software testing. In *14th International Conference on Quality Software*, pages 127–132, 2014.

[30] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 57–68, 2009.

[31] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, 2007.

[32] R. Pham, L. Singer, O. Liskin, K. Schneider, et al. Creating a shared understanding of testing culture on a social coding site. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 112–121, 2013.

[33] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, Oct 2001.

[34] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, 2010.

[35] I. Segall and R. Tzoref-Brill. Interactive refinement of combinatorial test plans. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1371–1374, 2012.

[36] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, 2005.

[37] S. Shahamiri, W. Kadir, and S. Mohd-Hashim. A comparative study on automated software test oracle methods. In *Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on*, pages 140–145, 2009.

[38] K. P. Singh, T. F. Bissyandé, D. Lo, L. Jiang, et al. An empirical study of adoption of software testing in open source projects. In *Proceedings of the 13th International Conference on Quality Software (QSIC 2013)*, pages 1–10, 2013.

[39] N. Tillmann and J. De Halleux. Pex–white box test generation for. net. In *Tests and Proofs*, pages 134–153. 2008.

[40] T. Williams, M. Mercer, J. Mucha, and R. Kapur. Code coverage, what does it mean in terms of quality? In *Reliability and Maintainability Symposium, 2001. Proceedings. Annual*, pages 420–424, 2001.

[41] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *ICSE*, pages 611–620, 2011.

[42] T. Xie, N. Tillmann, J. De Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 359–368, 2009.

[43] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, 2010.