Software Reversing Lab 1 Report

Kangqi Li 4518942
Yuzhu Yan 4468023

March 20, 2017

1 Introduction

In this report, we present two kinds of automated software testing techniques, fuzzing and concolic execution, for finding implementation bugs in programs. Two off-the-shell softwares, American Fuzzy Lop(AFL) and KLEE, which based on fuzzing and concolic execution techniques respectively, are introduced and compared with examples in this report. Subsequently, we deploy them on several Rigorous Examination of Reactive Systems (RERS) reachability problems for further investigation and comparison. Produced outputs are compared and analyzed at the end of this report.

2 Fuzzing

Fuzzing is an automated software testing technique for providing input of invalid, unexpected or random input to a computer program which is expected to trigger crashes, failing built-in code assertions or potential memory leaks due to bugs in programs. Fuzzing can be simply categorized into dumb fuzzing or smart fuzzing depending on whether it is aware of input structure, or generation- or mutation-based depending on whether inputs are generated from scratch of by modifying existing inputs, or white-, grey-, or blackbox depending on whether it is aware of program structure. Dumbing fuzzing, whose input is completely random with no knowledge of what the expected input should look like. It is an inexpensive method but with low efficiency in some situations. Another disadvantage of dumb fuzzing is that only certain inputs are accepted by program, in this case, Smart Fuzzing can be deployed as countermeasure. Smart fuzzing tool requires being programmed with knowledge of input format(A protocol definition or rules for a file format) for may create look like valid input with some alterations. Opposite to dumb fuzzing, smart fuzzing targets more precisely but with higher cost on designing.

The following code fragment is an example of explaining how fuzzing works:

```
int main(void){
    int input;
    scanf("%d", &input);
    if (input<10000 && input>1000){
        vulnerable();
    }
}
```

In this program, user provide an initial value, fuzzer proceed this value as input and keep mutating this value by following a certain rules or randomly. All generated values will be scanned into program and once the value is between 1000 and 10000, the program will raise a crash.

Despite fuzzer greatly reduce efforts for finding bugs which need to spend hours to find by manually, it still have many disadvantages. Firstly, bugs found by fuzzers are mostly very "shallow". Because some of them can only be triggered in highly specific circumstances. Random mutations make it very unlikely to reach certain code paths in the tested code, leaving some vulnerabilities firmly outside the reach of this technique[Ste16]. This also leads to very low efficient because it struggles to identify the precise values needed to satisfy checks on specific input. Secondly, crashing test cases generated by fuzzers could be difficult to analyze, since the act of fuzzing does not give you much knowledge of how the software operates internally.

3 Concolic Execution

The randomly concrete testing takes only one path each time based on the input value, which may leads to many paths becoming infeasible even with a huge amount of testing times. Therefore, the idea of symbolic testing is to execute programs using symbolic inputs rather than concrete values.[AT08] The tested programs should firstly be interpreted into abstract expressions. Each feasible path is identified and represented by certain logical formulas.[AT08] Therefore, a path is feasible when all its formulas are satisfiable and the searching for paths becomes a constraint satisfaction problem. For example, to check if the access of array **a[i]** can exceed the bound, the constraint problem will be to figure out if conjunction of path condition and $\mathbf{i<0} \lor \mathbf{i} > \mathbf{a.length}$ is satisfiable. When the execution find the current constraint is satisfiable with previous constraint, this constraint or its negation is added as a component of the entire constraint of this path then the execution proceeds to next state. Otherwise the current path terminates and the execution backtracks its previous state and move to another path with a reversed constraint. Take the following code as an example:

```
 1 int example(int a, int b, int c)
 2 {
 3 if (a >= b)
 4   if (a >= c) return a;
 5   else return c;
 6 else
 7   if (b >= c) return b;
 8   else
 9     if (a >= c) return a;
10 }
```

The function **example** aims to find the largest value of three inputs. The constraint formula for the first branch in line 3 is $\mathbf{(a >= b)}$, this constraint is satisfiable so the execution proceeds to line 4 which requires $\mathbf{(a >= c)}$. Since $\mathbf{(a >= b)}\lor\mathbf{(a >= c)}$ can be satisfied, the state is reachable and the path terminates with the return instruction. Then the execution retrieve back to line 4 and take the reversed constraint $\mathbf{(a < c)}$. Similar to the previous path, it is possible to satisfy the constraint. So the execution retrieve again for other paths. When the execution goes to line 9, the required constraint in previous states is $\mathbf{(a < b)}\land\mathbf{(b < c)}$. But the current constraint is $\mathbf{(a >= c)}$ which has a conflict with previous constraints. Hence, the following states of this path is not feasible. All concrete input values who meet the constraint will follow the corresponding path. In this way, each symbolic execution represents many exact program runs and the final coverage will become relatively high. Symbolic execution theoretically can take any feasible path. However, there are several drawbacks of symbolic execution. The number of feasible path will grow exponentially large with the increase of the program size.[Ste16] This path explosion problem will result in unlimited iterations and affect the testing efficiency.[Ste16] Moreover, as a kind of static analysis, symbolic execution is false positive because it does not practically execute the real tested programs .[Ste16] Furthermore, symbolic execution is a white-box technique which requires the knowledge of the source code (for abstract interpretion) and will encounter a large additional workload caused by the libraries referenced by the tested code.

Concolic testing is a hybrid software verification technique with symbolic execution.[SA05] The program will be executed concretely with a specific input. But the symbolic path conditions are collected along the execution. After one deterministic path is found and terminates, the constraint solver will analyze the collected constraints and negate another constraint for other paths. Then one of the concrete values which satisfies the new constraint will be the next concrete input value. The concolic testing iterates the execution with concrete input values derived from previous execution.[SA05] Hereby, the same as concrete testing, the analyzing targets of concolic testing are also the binaries. In this way, concolic testing is not false positive and does not require the knowledge of source codes. The path exploration can be relieved without affecting the coverage.

However, there are also some limitations of concolic testing. Compared with pure symbolic execution, concolic can significantly reduce the effect of path exploration. But it still cannot eliminate path exploration completely. And it could not reach the expected path when the tested program is nondeterministic, which means it cannot be applied on programs with random algorithms. When compared with symbolic execution and fuzzing, concolic execution is slow because of the time consumption for the code interpretation and the overhead involved in the constraint solving step.[Ste16] Moreover, John Regehr's blog shows an interesting example where KLEE, the concolic testing tool, can reach all paths but failed to find the error.[Reg11] This is because symbolic execution is very limited in what it can determine to be a bug. It easily failed to notice logical errors.

4 American Fuzzy Loop

AFL-fuzz is a brute-force fuzzer developed in C language that takes inputs from standard I/O approaches or a independent file. The logic of AFL is genetic fuzzing, which carries out input generation through a genetic algorithm, mutates inputs according to genetics inspired rules and ranks them by a fitness function. For AFL, the fitness function is based on the unique code coverage-that is, triggering an execution path that is different than the paths triggered by other inputs. The following figure1 depicts the general working approach of AFL.

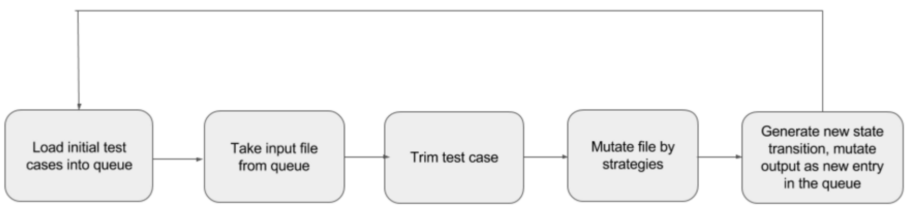

Load initial test cases into queue
Take input file from queue
Trim test case
Mutate file by strategies
Generate new state transition, mutate output as new entry in the queue

Figure 1: The general working approach of American Fuzzy Loop. User provides an initial test case to AFL. Then all the generated interesting test cases will be put into a queue. Fuzzer goes over the queue, does fuzzing, generates new entries, puts them into queue then loops back to the very beginning for next iterations.

As shown in Figure 1, the user is required to provide the initial test case for AFL fuzzer. It demands a read-only directory with initial test cases, a separate place to store its findings, plus a path to the binary to test. Program can be interrupted at any time by Ctrl-C. Output are stored in three directories: *queue*, *crashes* and *hangs*. Crashes contains unique test cases that cause the tested program to receive a fatal signal[1]. hangs includes all unique test cases that cause the tested program to time out. Once start running AFL-fuzzer, an UI appears at terminal, it shows shows how long it has been running, and how many unique code paths, crashes and hangs it has been able to find so far. Cycle resembles the queue of generated interesting test cases. Arithmetic is the complex of program. Once AFL detects a unique failure, it writes it into corresponding output directory.

The features and state-of-art black box genetic algorithm of AFL makes it a powerful fuzzing tool, and it even can handle loop problems which is complicated for other fuzzing engines and concolic execution engines by its loop "bucketization" mechanism[Ste16]. However, it also has some well-known limitations. The generation of "specific" input to pass complex checks in application is very challenging for AFL, since the mutating input has a very small chance of sending the correct input. Such circumstance leads to a small chance of successfully shooting target paths once any highly specific circumstances are required.

[1]Sometimes it is hard to detect crash. This could be explained by: afl-fuzz distinguishes between hangs and crashes by waiting for a timeout to expire. A crash is only reported as a crash when it is reached before time out, otherwise it is a successful run. In some operating system, like OS X, generating crash report always takes substantially longer than afl-fuzz's default timeout of 20 milliseconds. So crashes can barely see. However, user can configure timeout by flag -t

5 KLEE

KLEE is a symbolic execution tool which can automatically generate tests for various programs with an average coverage over 90%.[CE08] It is capable to find deep bugs in complex system programs without the requirements for source modifications or manual work.[CE08]
Before testing execution, users firstly need to compile the source code to bytecode by LLVM compiler. KLEE automatically interprets the instructions into abstract constrains. The core of KLEE is an interpreter loop which selects a state to run and then symbolically executes a single instruction in the context of that state. This loop continues until there are no states remaining, or an userdefined timeout is reached.[CE08]
Compared with other concolic executions, KLEE has several features which can improve the efficiency, coverage and correctness. Algorithm is implemented to select path randomly. When a branch point is reached, the set of states in each subtree has equal probability of being selected, regardless of the size of their subtrees.[CE08] This strategy avoids starvation when some part of the program is rapidly creating new states and it favors states high in the branch tree which have less constraints on their symbolic inputs and so have greater freedom to reach uncovered code.[CE08]
Coverage-Optimized Search tries to select states likely to cover new code in the immediate future. It uses heuristics to compute a weight for each state and then randomly selects a state according to these weights. KLEE uses each strategy in a round robin fashion.[CE08]

6 Experiments

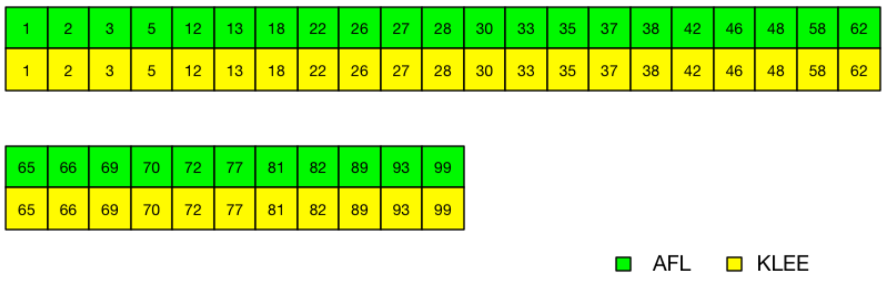
6.1 RERS Reachability Challenge

Rigorous Examination of Reactive Systems(RERS) challenges provide a forum of experimental profile evaluation based on specifically designed Benchmark suites as a countermeasure for comparing multiple validation techniques of reactive systems. In this report, we test AFL and KLEE separately for several problems of RERS Challenges. By comparing reached errors and execution conditions, we represent our investigation of these two testing tools.

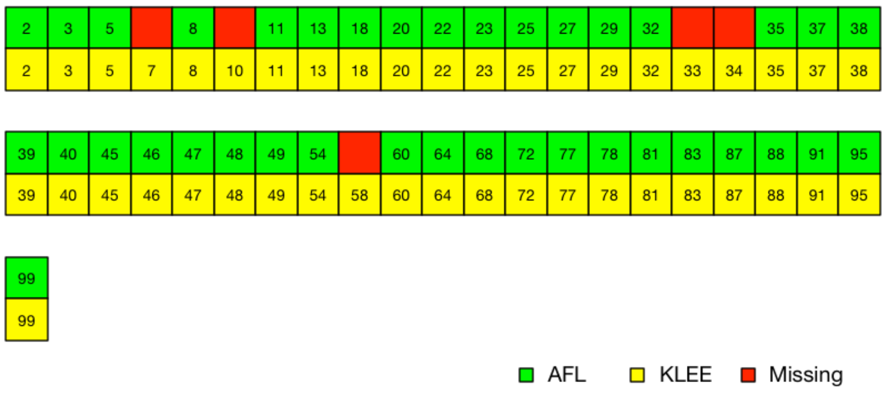
6.2 Result and Analysis

We select three problems of 2017 RERS reachability problems, Problem10, Problem14 and Problem15, where Problem10 is plain and with a small size, Problem14 is arithmetic and in medium size, Problem15 has a complex data structure and a medium size. These three problems are simple, medium and hard for analysis respectively. The initial input value for AFL is 1. And the execution is terminated after 1 hour.

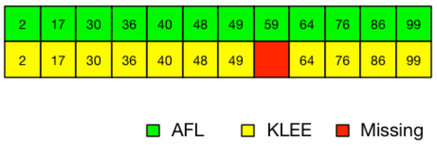
Figure 2 shows the IDs of reached errors. Besides the output of reached errors, there are always one assertion error and one execution error found by KLEE for all reachability problem. As the simplest problem, both AFL and KLEE succeeded in reaching 32 errors in Problem 10. Figure 3 depicts three AFL outputs for corresponding problems. Figure 4 shows the conditions of three testings of KLEE. For KLEE, although the testing times are different, we can still find that it can reach the same errors within approximately 30 minutes, which is the half execution time of AFL. But we cannot find the exact time when AFL find all errors because we always terminate AFL manually after 1 hour execution each time. Therefore, it is hard to compare the efficiency of these two methods. It also shows that the instruction coverage and branch coverage of KLEE execution are 92.66% and 83.91% respectively. In Figure 3, the Path geometry column, "levels" represents the path depth reached through the guided fuzzing process while "pending" is the number of inputs that have not gone through any fuzzing yet. Figure 3a shows a path depth level of 12 and only 364 pending inputs in current mutation. Moreover, in "overall result" column, there are 2 cycles finished. Therefore, we conclude the AFL is capable to give a high coverage of the entire program and the found errors are expected to be all the reachable errors of Problem 10 and both KLEE and AFL can solve this problem with an acceptable time consumption. In addition, at "fuzzing strategy yields" block in figure3a, you can also see arithmetics is 27/498k, however it dramatically increases as the increasing of complexity of program, which shows as 166/539k and 204/869k for Problem 14 and 15.



(a) Problem 10



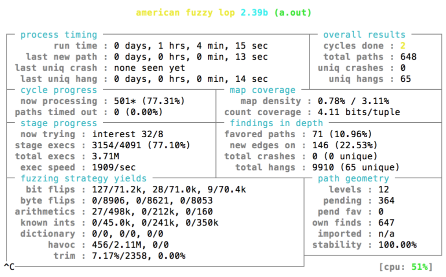
(b) Problem 14



(c) Problem 15

Figure 2: The comparison of reached errors by running AFL and KLEE.

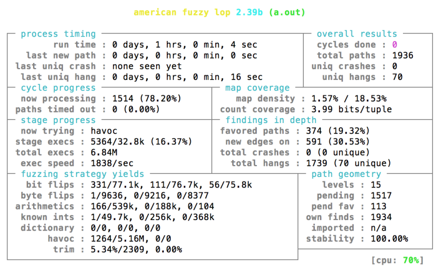
Figure 2b shows there are 43 errors reached by KLEE and 38 errors reached by AFL which represents that 1 hour is far away from enough for AFL to find all results. Figure 4b suggests that as the total number of instructions decreases, the execution time of Problem 14 is much less than that of Problem 10. An assumption is that the time spent on interpreting instructions into abstract constraints gets smaller. However, as the complexity grows, the instruction coverage and the branch coverage become less than half of that in Figure 4a. And the percentage of time spent on the constraint solver becomes larger. The reason may be the path exploration problem limits KLEE to find new paths and permits KLEE to find deeper states and takes the constraint solver a large time to target new states. For the performance of AFL, there are 5 reachable errors not found within 1 hour. Figure 3b shows the total covered paths is 1936, which is more than the twice of that in Figure 3a and there is no cycle finished yet. Hence the number of the entire paths of Problem 14 are very large. The level of path depth in Figure 3b is 15, which is larger than that in 3a. This also depicts the complexity of Problem 14. Considering the output of printed errors in the execution of KLEE, error_7, error_10 and error_33 appeared only once each time while error_34 and error_58 appeared only twice for each KLEE execution. Our explanation towards this phenomena is these errors can only be reached by only a few specific input values. Since AFL chooses input values randomly and even in a depth of level 15, there are still thousands of pending inputs waiting for execution, there is a poor probability of AFL to shoot the specific input values and reach the corresponding paths. But it is possible of KLEE to find them unless these errors show up in very deep states. Moreover, AFL could not finished one entire cycle in one hour(figure3b), which means fuzzer did not go over all interesting test cases discovered, so AFL is



```
                    american fuzzy lop 2.39b (a.out)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 1 hrs, 4 min, 15 sec        │  cycles done : 2      │
│   last new path : 0 days, 0 hrs, 0 min, 13 sec        │  total paths : 648    │
│ last uniq crash : none seen yet                       │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 0 min, 14 sec        │   uniq hangs : 65     │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 501* (77.31%)     │    map density : 0.78% / 3.11%         │
│ paths timed out : 0 (0.00%)         │ count coverage : 4.11 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : interest 32/8         │ favored paths : 71 (10.96%)            │
│ stage execs : 3154/4091 (77.10%)    │  new edges on : 146 (22.53%)           │
│ total execs : 3.71M                 │ total crashes : 0 (0 unique)           │
│  exec speed : 1909/sec              │   total hangs : 9910 (65 unique)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 127/71.2k, 28/71.0k, 9/70.4k          │    levels : 12         │
│  byte flips : 0/8906, 0/8621, 0/8053                │   pending : 364        │
│ arithmetics : 27/498k, 0/212k, 0/160                │  pend fav : 0          │
│  known ints : 0/45.0k, 0/241k, 0/350k               │ own finds : 647        │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 456/2.11M, 0/0                        │ stability : 100.00%    │
│        trim : 7.17%/2358, 0.00%                     ├────────────────────────┘
^C────────────────────────────────────────────────────┘             [cpu: 51%]
```

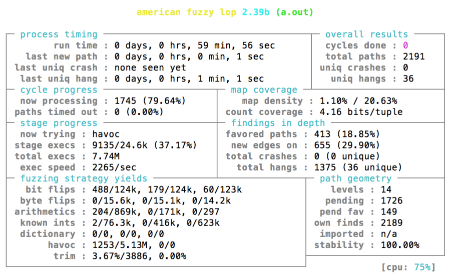
(a) Problem10



```
                    american fuzzy lop 2.39b (a.out)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 1 hrs, 0 min, 4 sec         │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 0 sec         │  total paths : 1936   │
│ last uniq crash : none seen yet                       │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 0 min, 16 sec        │   uniq hangs : 70     │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 1514 (78.20%)     │    map density : 1.57% / 18.53%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 3.99 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 374 (19.32%)           │
│ stage execs : 5364/32.8k (16.37%)   │  new edges on : 591 (30.53%)           │
│ total execs : 6.84M                 │ total crashes : 0 (0 unique)           │
│  exec speed : 1838/sec              │   total hangs : 1739 (70 unique)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 331/77.1k, 111/76.7k, 56/75.8k        │    levels : 15         │
│  byte flips : 1/9636, 0/9216, 0/8377                │   pending : 1517       │
│ arithmetics : 166/539k, 0/188k, 0/104               │  pend fav : 113        │
│  known ints : 1/49.7k, 0/256k, 0/368k               │ own finds : 1934       │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 1264/5.16M, 0/0                       │ stability : 100.00%    │
│        trim : 5.34%/2309, 0.00%                     ├────────────────────────┘
└─────────────────────────────────────────────────────┘             [cpu: 70%]
```

(b) Problem14

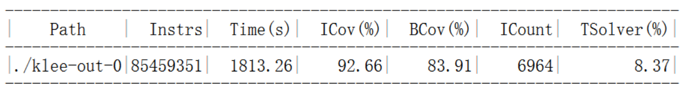


```
                    american fuzzy lop 2.39b (a.out)

┌─ process timing ─────────────────────────────────────┬─ overall results ─────┐
│        run time : 0 days, 0 hrs, 59 min, 56 sec       │  cycles done : 0      │
│   last new path : 0 days, 0 hrs, 0 min, 1 sec         │  total paths : 2191   │
│ last uniq crash : none seen yet                       │ uniq crashes : 0      │
│  last uniq hang : 0 days, 0 hrs, 1 min, 1 sec         │   uniq hangs : 36     │
├─ cycle progress ────────────────────┬─ map coverage ─┴───────────────────────┤
│  now processing : 1745 (79.64%)     │    map density : 1.10% / 20.63%        │
│ paths timed out : 0 (0.00%)         │ count coverage : 4.16 bits/tuple       │
├─ stage progress ────────────────────┼─ findings in depth ────────────────────┤
│  now trying : havoc                 │ favored paths : 413 (18.85%)           │
│ stage execs : 9135/24.6k (37.17%)   │  new edges on : 655 (29.90%)           │
│ total execs : 7.74M                 │ total crashes : 0 (0 unique)           │
│  exec speed : 2265/sec              │   total hangs : 1375 (36 unique)       │
├─ fuzzing strategy yields ───────────┴───────────────┬─ path geometry ────────┤
│   bit flips : 488/124k, 179/124k, 60/123k           │    levels : 14         │
│  byte flips : 0/15.6k, 0/15.1k, 0/14.2k             │   pending : 1726       │
│ arithmetics : 204/869k, 0/171k, 0/297               │  pend fav : 149        │
│  known ints : 2/76.3k, 0/416k, 0/623k               │ own finds : 2189       │
│  dictionary : 0/0, 0/0, 0/0                         │  imported : n/a        │
│       havoc : 1253/5.13M, 0/0                       │ stability : 100.00%    │
│        trim : 3.67%/3886, 0.00%                     ├────────────────────────┘
└─────────────────────────────────────────────────────┘             [cpu: 75%]
```

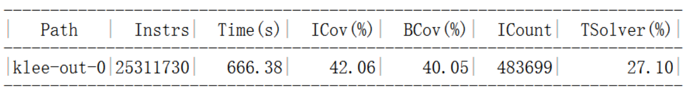
(c) Problem15

Figure 3: AFL testing output of RERS challenges.



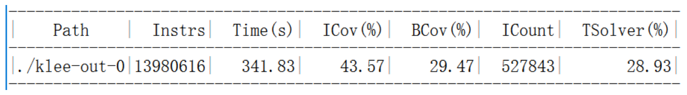
Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
./klee-out-0	85459351	1813.26	92.66	83.91	6964	8.37

(a) Problem 10



Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
klee-out-0	25311730	666.38	42.06	40.05	483699	27.10

(b) Problem 14



Path	Instrs	Time(s)	ICov(%)	BCov(%)	ICount	TSolver(%)
./klee-out-0	13980616	341.83	43.57	29.47	527843	28.93

(c) Problem 15

Figure 4: The testing analysis of KLEE for RERS challenges. Instrs: number of executed instructions; Time: total wall time (s); ICov: instruction coverage in the LLVM bitcode (%); BCov: branch coverage in the LLVM bitcode (%); ICount: total static instructions in the LLVM bitcode; TSolver: time spent in the constraint solver

also significantly limited by this execution time in this case. [2]
As for the Problem 15, which is the most difficult problem we tested, Figure 2c suggests that only 11 errors are found and error_59 is missed by KLEE. In the consideration of Figure 2b, the reason may be the state of error_59 is too deep to be found by KLEE. Figure 4c also proves that the path exploration extremely limits the performance of KLEE for this problem as the branch coverage is only 29.47% and execution time is around 340 seconds. KLEE is not capable to reach new paths anymore because of the insufficient power of the constraint solver. But if the error can be triggered by various input values, which means there is no highly specific circumstance required, it is possible for AFL to randomly selected the specific values and reach the corresponding errors within 1 hour. This probably can explain the different performance between AFL and KLEE in Problem 15.

[2]AFL official tutorial suggests that "Every fuzzing session should be allowed to complete at least one cycle; and ideally, should run much longer than that" Source: http://lcamtuf.coredump.cx/afl/status_screen.txt

7 Conclusion

Since we cannot get the answer of these three problems and it is impossible to analyze the exact structure and branches of them, our conclusion is mainly derived from the results and the features of AFL and KLEE. We think KLEE works well for simple and small programs and can discover an acceptable amount of feasible path efficiently. But it is hard to explore complex programs with a large amount of branches and deep states. The coverage of AFL are affected by the execution time a lot. Once the feasible path can only be reached by a highly specific circumstance, in our case is a small amount of specific input values, it would be very hard for AFL to find them within a limited time.

References

[AT08] Patrice Godefroid Anand, Saswat and Nikolai Tillmann. Demand-driven compositional symbolic execution. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.*, pages 367–381, 2008.

[CE08] Daniel Dunbar Cadar, Cristian and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. *OSDI*, 8:209–224, 2008.

[Reg11] John Regehr. A few thoughts about path coverage. *http://blog.regehr.org/archives/386*, 2011.

[SA05] Darko Marinov Sen, Koushik and Gul Agha. Cute: a concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes.*, 30(5):263–272, 2005.

[Ste16] et al. Stephens, Nick. Driller: Augmenting fuzzing through selective symbolic execution. *Proceedings of the Network and Distributed System Security Symposium*, 2016.