Characterizing and Improving Bug-Finders with Automated Bug Injection

Yu Hu, Zekun Shen,
Brendan Dolan-Gavitt
06/10/2021

Large-scale Automated Vulnerability Addition (LAVA)

- LAVA is an automatic tool that can inject multiple bugs to programs through source-level instrumentation powered by LLVM
- Ground-truth vulnerability corpora database

Features

- Known information about these bugs
  - Number of bugs
  - Types and locations
- Bugs come with triggering inputs
- Inject bugs in real world program!

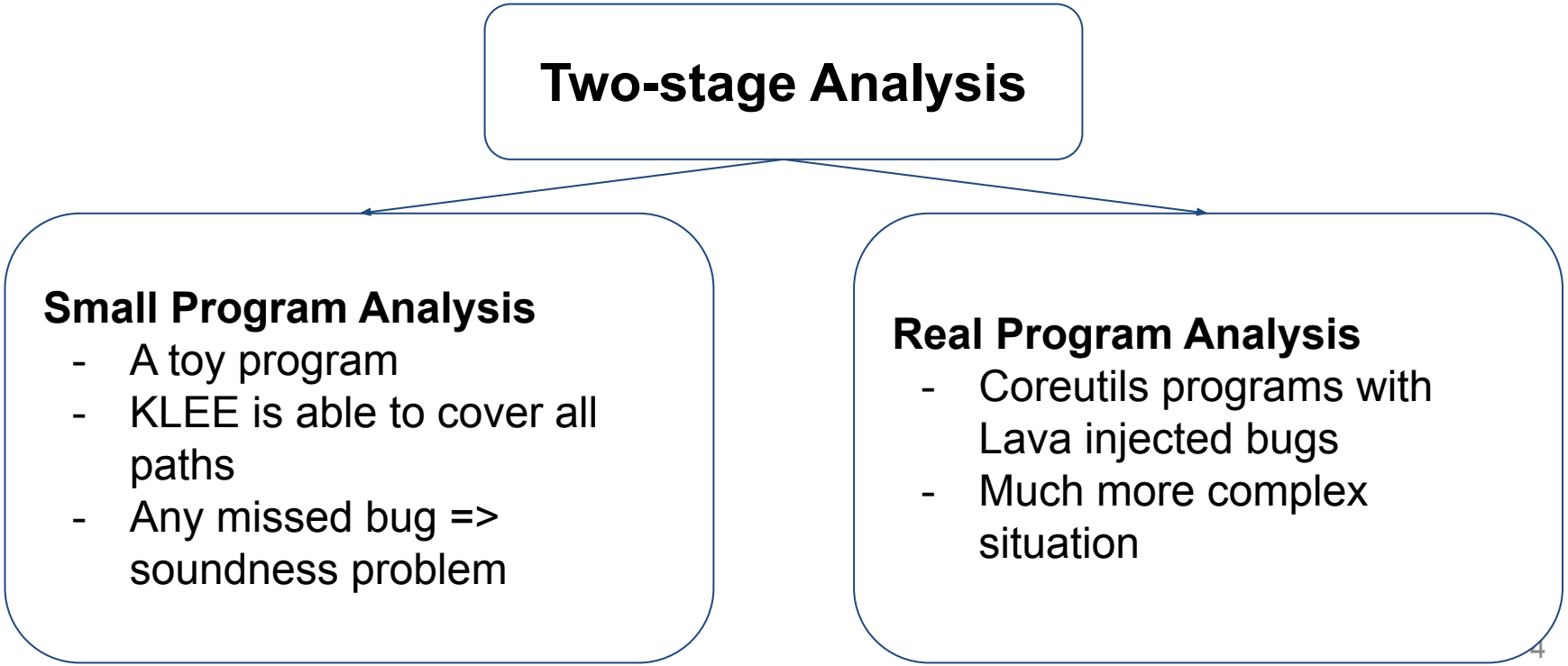
Background

This lets us investigate interesting questions about KLEE empirically:

- When might KLEE miss a bug?
- What's the effect of symbolic file size?
- What search strategies work best for bug-finding?
- Relationship between coverage and bugs found?
- Does the depth of injected bugs affect KLEE's bug finding performance?

Methodology

RQ: When might KLEE miss a bug?



Methodology

Small program analysis with LAVA injected bugs

- ***Toy.c*** with only 70 lines, input is a binary file, output is meta information
- 159 buggy program produced
- Injected into the parameter of call to user-defined function and external functions
- 5 hours running KLEE, with sufficient resources
- Never expire, never kill the states, KLEE exits normally
- Probe the soundness problem

Small Program Evaluation

Evaluation - Small Program

- 97% instruction coverage and 100% branch coverage
- 3% missed instructions are all Exit(1) for fail to read the file

Function Category	KLEE v1.4	KLEE v1.4 with printf enabled	KLEE v2.2	Total bugs
user-defined function	68	68	68	72
printf function	0	31	82	87
Total	68	99	150	159

Small Program Evaluation

Evaluation - Small Program

Function Category	KLEE v1.4	KLEE v1.4 with printf enabled	KLEE v2.2	Total bugs
user-defined function	68	68	68	72
printf function	0	31	82	87
Total	68	99	150	159

KLEE can’t support floating point

Small Program Evaluation

Evaluation - Small Program

Function Category	KLEE v1.4	KLEE v1.4 with printf enabled	KLEE v2.2	Total bugs
user-defined function	68	68	68	72
printf function	0	31	82	87
Total	68	99	150	159

KLEE v1.4 does not check pointers passed to external library calls for validity

Small Program Evaluation

Example of the code

```
1 if (ent->type == TYPEA) {
2     printf("fdata = %f\n" \
3         +(lava_get(128))*(0x6c6175e1==(lava_get(128)) \
4         ||0xe175616c==(lava_get(128))), ent->data.fdata);
5 }
```

Real Program Evaluation

When might KLEE miss a bug?

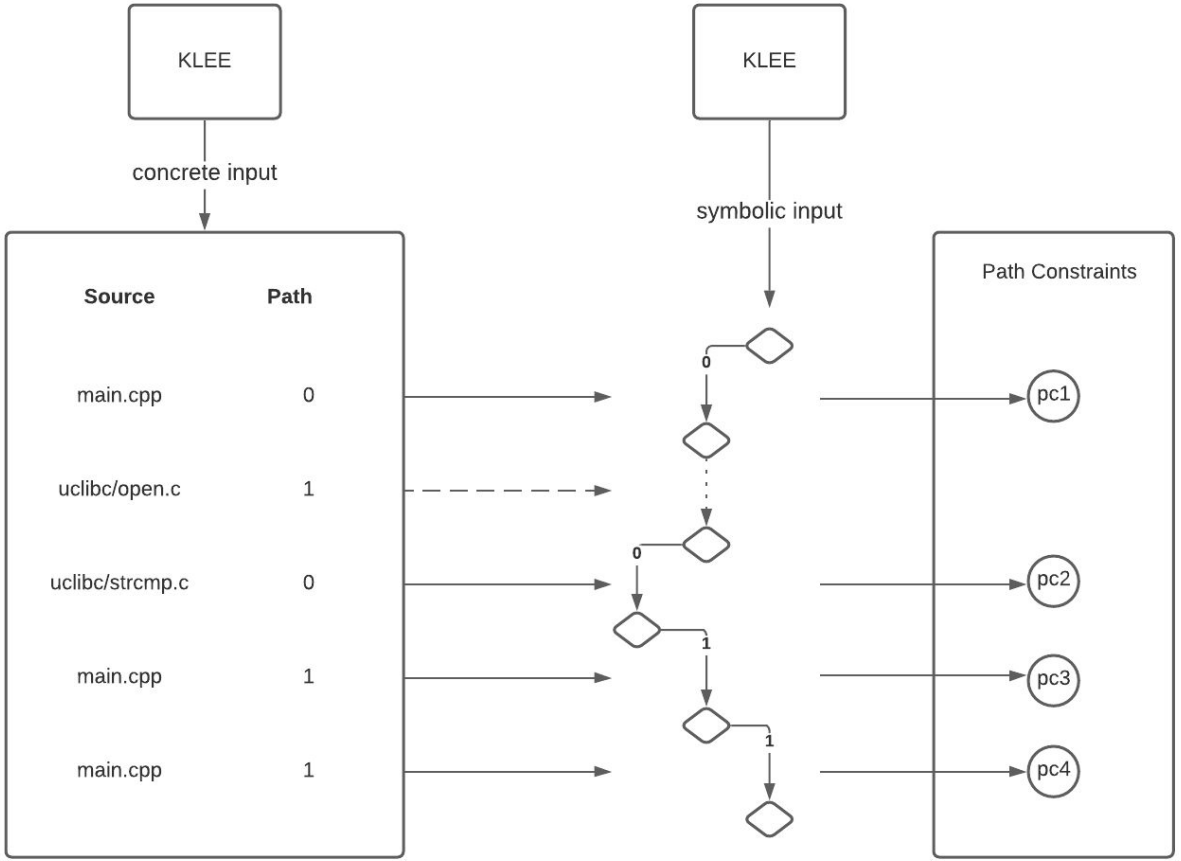
- It can be difficult to distinguish false negatives caused by resource limitations (e.g., timeouts, insufficient symbolic input, path explosion problem) from actual soundness problems.

- Concolic execution to the rescue!

Real Program Evaluation

Concolic KLEE

We force KLEE to follow the path taken by an input known to trigger the bug, which factors out the difficulty of path search from the bug-finding task.



Real Program Evaluation

Real program analysis results

- 2000+ bugs, selected 243 with unique attack point

	Injected Bugs	Found (v1.4)	Found (v2.2)	Found (Concolic)	uClibc Compiled?	Triggered (uClibc)
base64	28	22	20	28	yes	28
cat	8	6	6	7	yes	7
cut	3	1	1	2	no	-
od	24	9	13	22	yes	22
pr	2	0	0	0	yes	0
ptx	20	3	3	20	yes	20
sum	7	1	1	6	no	-
tail	8	3	6	8	yes	8
uniq	12	6	6	1	no	-
who	119	5	5	5	yes	5
unexpand	4	0	1	4	no	-
head	1	0	1	1	yes	1
nl	3	3	3	3	no	-
expand	4	0	0	4	no	-

Real Program Evaluation

Why did Concolic KLEE miss these bugs?

	Injected Bugs	Found (v1.4)	Found (v2.2)	Found (Concolic)	uClibc Compiled?	Triggered (uClibc)
base64	28	22	20	28	yes	28
cat	8	6	6	7	yes	7
cut	3	1	1	2	no	-
od	24	9	13	22	yes	22
pr	2	0	0	0	yes	0
ptx	20	3	3	20	yes	20
sum	7	1	1	6	no	-
tail	8	3	6	8	yes	8
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
who	119	5	5	5	yes	5
unexpand	4	0	1	4	no	-
head	1	0	1	1	yes	1
nl	3	3	3	3	no	-
expand	4	0	[illegible]	[illegible]	[illegible]	-

who missed 114 out of 119

Real Program Evaluation

Why Concolic KLEE missed these bugs?

- Differences between glibc and KLEE's uClibc.
  - Some features missed in uClibc, e.g, libio.h
  - File Structure differences
- **Example:**

```
struct GLIBC_File {
    int nbytes; // offset 0
    char* _IO_write_base; // 4
    char* _IO_read_base; // 12
}
```

```
struct UCLIBC_File {
    int nbytes; // offset 0
    size_t max_size; // 4
    char* first_byte; // 12
}
```

Real Program Evaluation

Result of Concolic KLEE matches the number of correct triggered bug with uClibc

	Injected Bugs	Found (v1.4)	Found (v2.2)	Found (Concolic)	uClibc Compiled?	Triggered (uClibc)
base64	28	22	20	28	yes	28
cat	8	6	6	7	yes	7
cut	3	1	1	2	no	-
od	24	9	13	22	yes	22
pr	2	0	0	0	yes	0
ptx	20	3	3	20	yes	20
sum	7	1	1	6	no	-
tail	8	3	6	8	yes	8
uniq	12	6	6	1	no	-
who	119	5	5	5	yes	5
unexpand	4	0	1	4	no	-
head	1	0	1	1	yes	1
nl	3	3	3	3	no	-
expand	4	0	0	4	no	-

Real Program Evaluation

Research Questions:

- What's the effect of symbolic file size?
- What search strategies work best for bug-finding?
- Relationship between coverage and bugs found?
- Does the depth of injected bugs affect KLEE's bug finding performance?

Following examples are based on base64

Real Program Evaluation

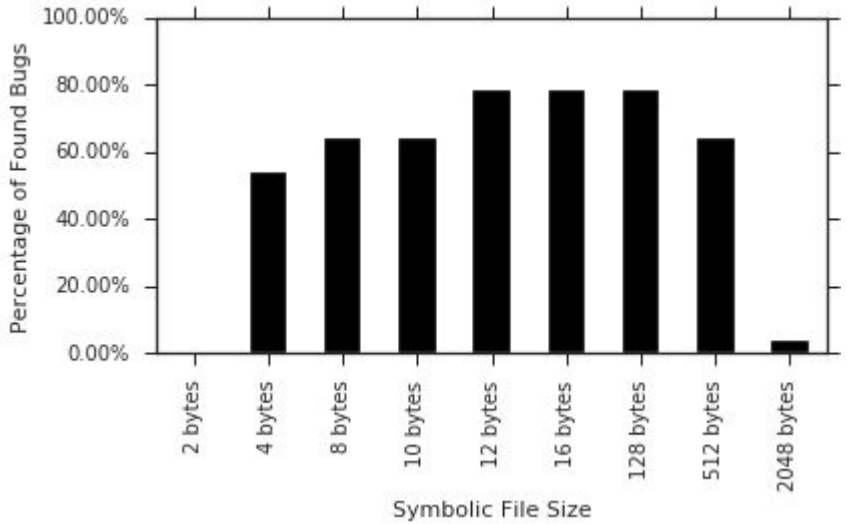
Symbolic File Size

Motivation:

- KLEE uses symbolic input file to replace concrete input file

Results:

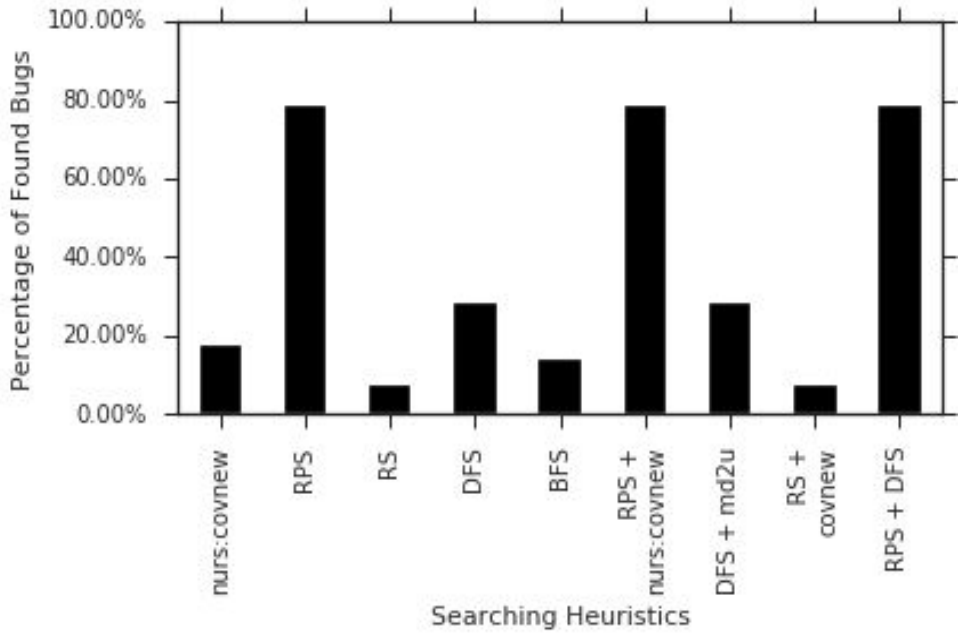
- Symbolic file size significantly impact found bugs
- Minimum symbolic file size is 4
- 12 bytes, 16 bytes and 128 bytes have best performance



Real Program Evaluation

Searching Heuristics

- LAVA bug contains lava_set and lava_get
- Random Path Selection outperforms all other searching heuristics
- The bug-finding rate of interleaved searching heuristics based on the best searching heuristics, except random state search



Real Program Evaluation

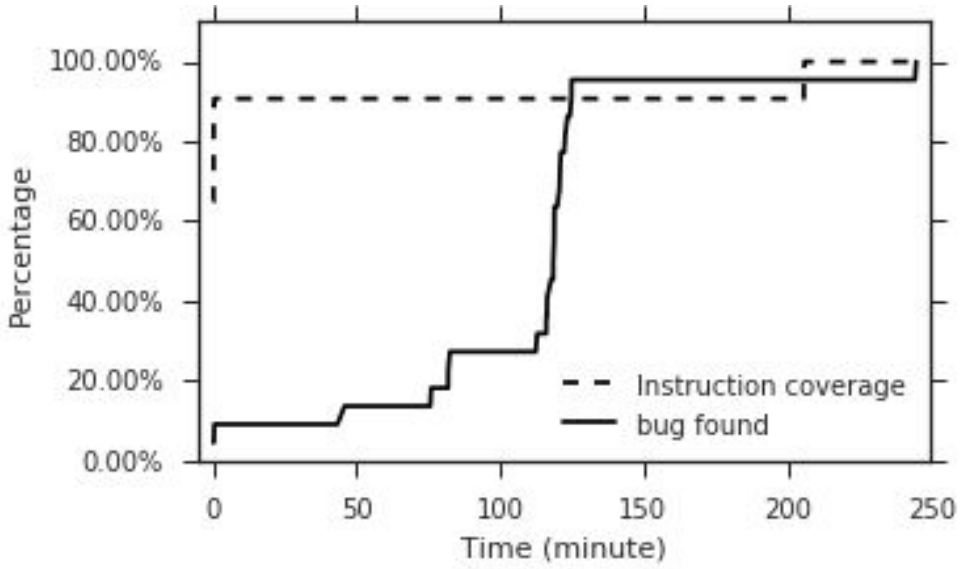
Instruction coverage

Motivation:

- Is line coverage correlated with the number of finding bugs?

Results:

- Limitation of max found bugs
- Not directly correlated to the bug-finding rate



Real Program Evaluation

Depth of injected bugs

Motivation

- What are “deep” bugs?
- How the “deep” bugs correlate to the bug-finding?

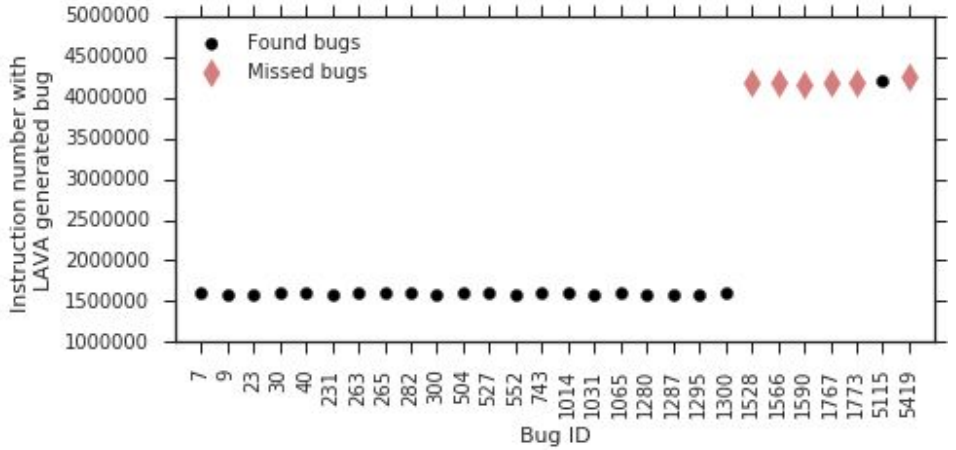
Definition of “Depth”

- In our experiment, “Depth” is not as simple as computing the distance from the start of the program to the point where a bug manifests

Real Program Evaluation

Depth of injected bugs

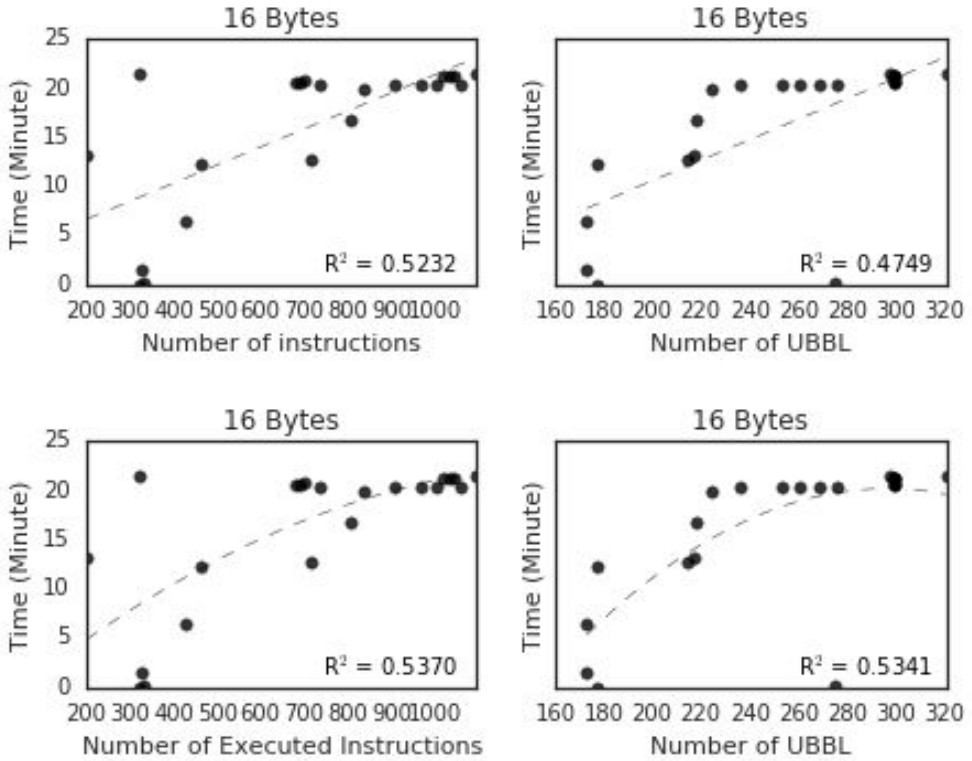
- Using ***perf*** to measure the number of instructions executed for each bug with the trigger input generated by LAVA
- The result shows 7 out of 28 bugs are deeply injected by LAVA
- KLEE found all 21 "shallow" bugs, but only found one out of the seven deeply injected bugs.



Real Program Evaluation

Depth of injected bugs example

- **Static analysis:** shortest path vs. finding bugs
- **Dynamic analysis:** number of unique basic blocks(UBBL) vs. finding bugs



Conclusion

Main goal: To see whether automatically generated bugs are actually useful for understanding, evaluating, and improving bug-finding tools.

Small program analysis: evaluating the soundness issues, floating point, printf
Real program analysis: let us understand how a bug-finding tool performs under different configurations on real-world programs.
Concolic KLEE: our main finding from this experiment is that discrepancies between uClibc (KLEE's libc implementation) and glibc can cause KLEE to miss some bugs or report errors that are hard to reproduce.