KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs

Cristian Cadar, Daniel Dunbar, Dawson Engler

Presented by Changfeng Liu, Jiachen Sun, and Shengtuo Hu

Problems

- Code complexity
  - *Non-obvious input parsing code*
  - *Tricky boundary conditions*
  - *Hard-to-follow control flow*
- Environmental dependencies (e.g., OS, network)
  - *Complex interactions with environmental input (including malicious input)*

What is KLEE?

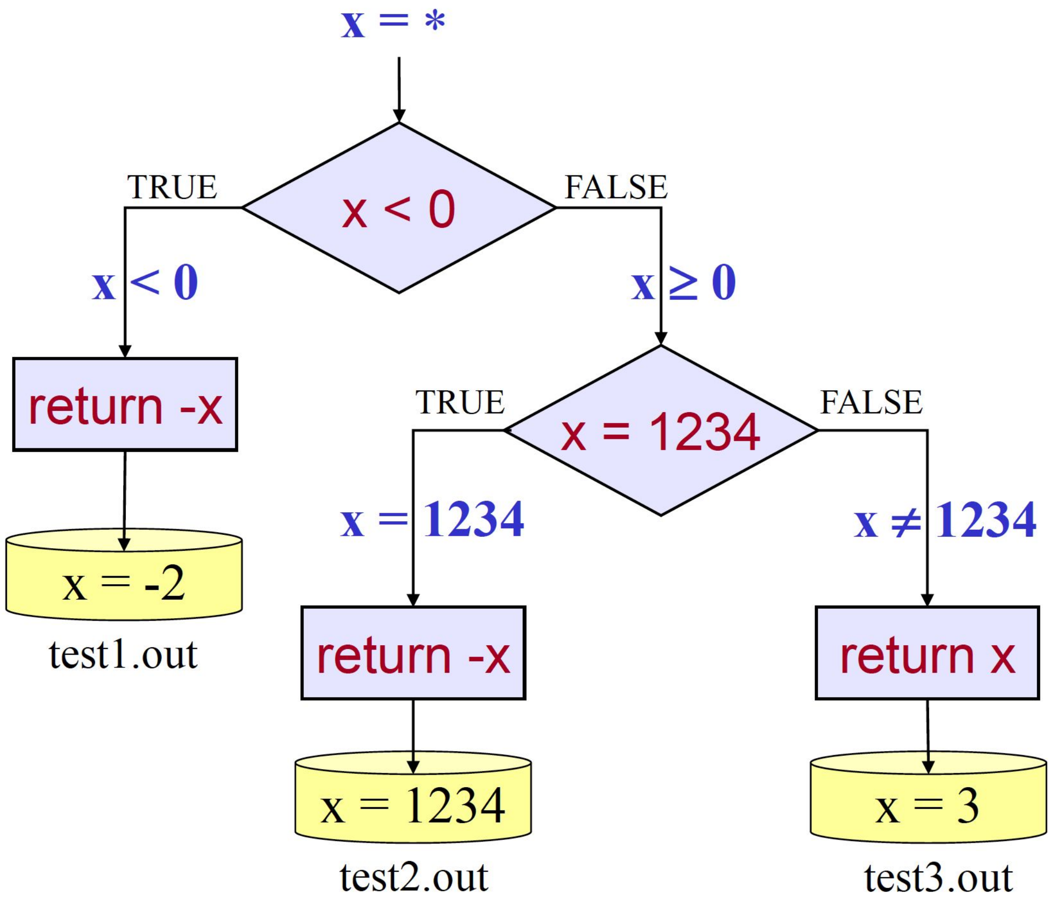
- Symbolic execution
- Constraint solving

It can:

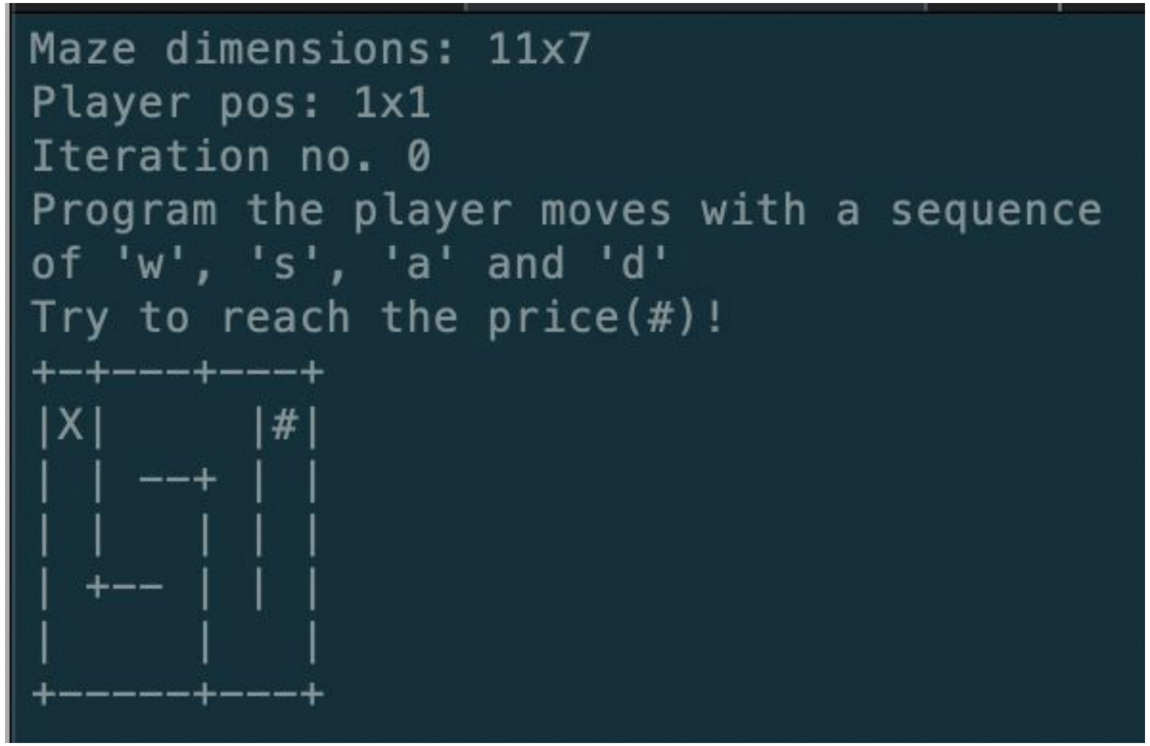
- Automatically generate high coverage test suites
- Find deep bugs in complex system programs

Example

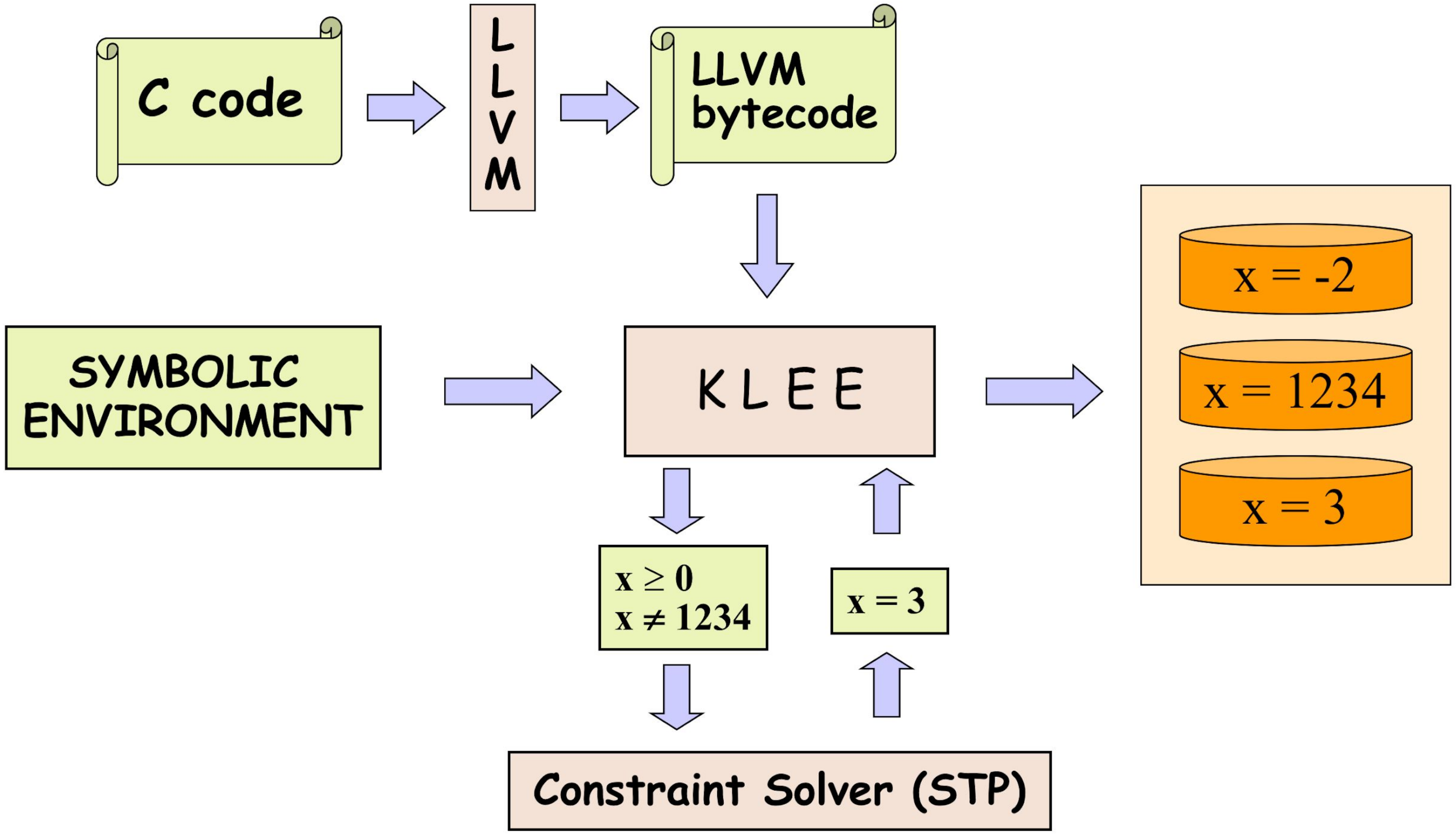
```
int bad_abs(int x)
{
    if (x < 0)
        return -x;
    if (x == 1234)
        return -x;
    return x;
}
```



Example - Maze



KLEE Architecture



Challenges

- State explosion
- Path selection
- Constraint solving
- Environment problem

State Explosion

Problem: the number of states grows very quickly and use tons of memory

Use compact state representation:

- Copy-on-write at the object level rather than page level
- Heap as an immutable map can be partially shared among states
- Heap can be cloned in constant time

Path Selection

Problem: selecting a path at random can easily get stuck

Use search heuristics:

- Random path selection
- Coverage-optimized search

Constraint Solving Optimization

Problem: the cost of constraint solving dominates runtime

Two types of optimizations:

- Eliminating irrelevant constraints
- Caching solutions

Eliminating Irrelevant Constraints

Each branch usually depends on a small number of variables

Example:

- **Constraint set**: {i < j, j < 20, k > 0}
- **Query**: i = 20 ?

Eliminating Irrelevant Constraints

Each branch usually depends on a small number of variables

Example:

- **Constraint set**: {i < j, j < 20, ~~k > 0~~}
- **Query**: i = 20 ?

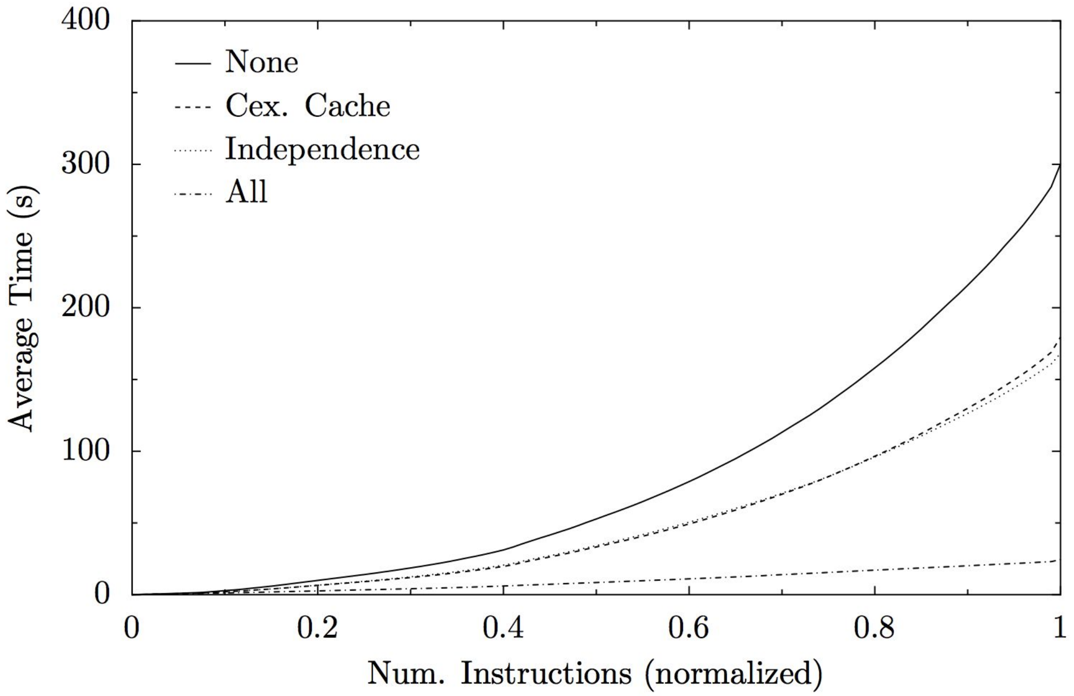
Caching Solutions

Example:

Cached entries	New queries
*{**i < 10, i = 10**} => no solution*	*{**i < 10, i = 10**, j = 12} => no solution*
*{**i < 10, j = 8**} => satisfiable, with i = 5, j = 8*	*{**i < 10**} or {**j = 8**} => satisfiable, with i = 5, j = 8*
*{**i < 10, j = 8**} => satisfiable, with i = 5, j = 8*	*{**i < 10, j = 8**, i != 3} => satisfiable, with i = 5, j = 8*

Optimization results

Optimizations	Queries	Time (s)	STP Time (s)
None	13717	300	281
Independence	13717	166	148
Cex. Cache	8174	177	156
All	699	20	10



Environment Modeling

Problem: interactions with the environment are complex

A hybrid solution:

- Forward concrete system calls to OS
- Handle function calls with symbolic arguments with *models*

Evaluation: In-depth Coverage Experiments

Methodology:

- Run KLEE one hour per utility in COREUTILS and BUSYBOX to generate test cases
- Run test cases
- Measure line coverage using **gcov**

Evaluation: Coverage Results (COREUTILS)

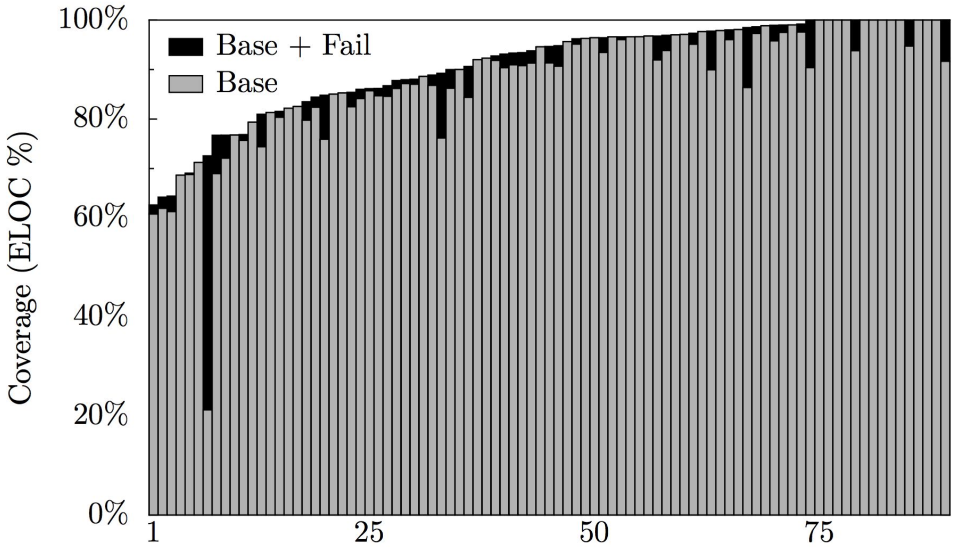


Figure 5: Line coverage for each application with and without failing system calls.

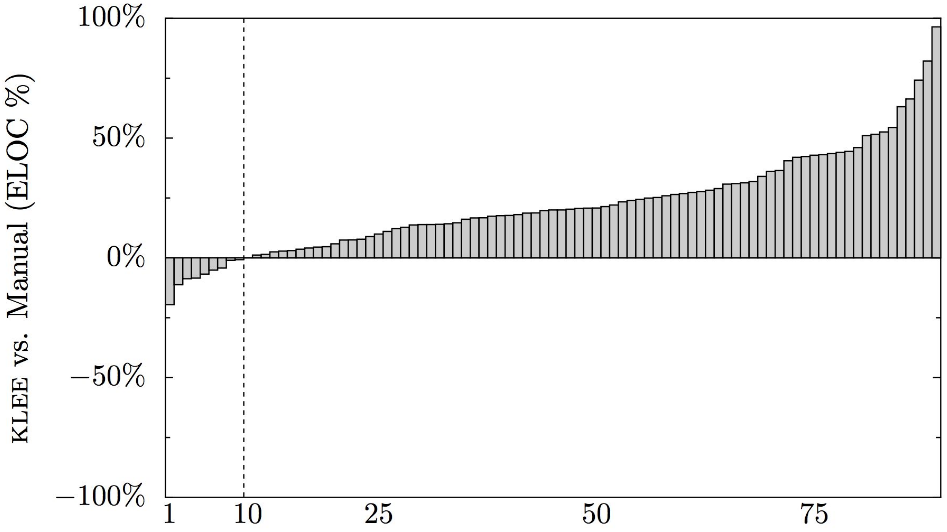
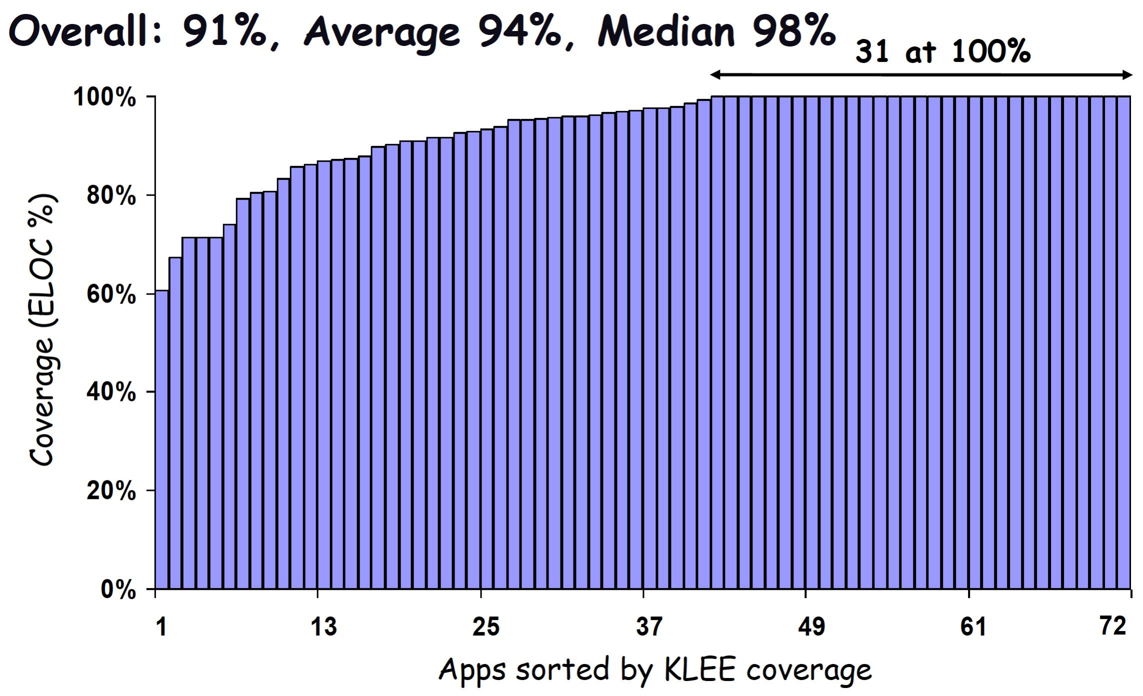


Figure 6: Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests (L_{man}) from KLEE tests (L_{klee}) and dividing by the total possible: $(L_{klee} - L_{man})/L_{total}$. Higher bars are better for KLEE, which beats manual testing on all but 9 applications, often significantly.

Evaluation: Coverage Results (BUSYBOX)

Coverage (w/o lib)	COREUTILS KLEE tests	COREUTILS Devel. tests	BUSYBOX KLEE tests	BUSYBOX Devel. tests
100%	16	1	31	4
90-100%	40	6	24	3
80-90%	21	20	10	15
70-80%	7	23	5	6
60-70%	5	15	2	7
50-60%	-	10	-	4
40-50%	-	6	-	-
30-40%	-	3	-	2
20-30%	-	1	-	1
10-20%	-	3	-	-
0-10%	-	1	-	30
Overall cov.	84.5%	67.7%	90.5%	44.8%
Med cov/App	94.7%	72.5%	97.5%	58.9%
Ave cov/App	90.9%	68.4%	93.5%	43.7%



Evaluation: Bug Finding (COREUTILS)

- 10 crash bugs

```
paste -d\\ abcdefghijklmnopqrstuvwxyz
pr -e t2.txt
tac -r t3.txt t3.txt
mkdir -Z a b
mkfifo -Z a b
mknod -Z a b p
md5sum -c t1.txt
ptx -F\\ abcdefghijklmnopqrstuvwxyz
ptx x t4.txt
seq -f %0 1
```

t1.txt: `"\t \tMD5("`
t2.txt: `"\b\b\b\b\b\b\b\t"`
t3.txt: `"\n"`
t4.txt: `"a"`

Evaluation: Bug Finding (BUSYBOX)

- 21 crash bugs

```
date -I
ls --co
chown a.a -
kill -l a
setuidgid a ""
printf "% *" B
od t1.txt
od t2.txt
printf %
printf %Lo
tr [
tr [=
tr [a-z
```

```
t1.txt: a
t2.txt: A
t3.txt: \t\n
```

```
cut -f t3.txt
install --m
nmeter -
envdir
setuidgid
envuidgid
envdir -
arp -Ainet
tar tf_ /
top d
setarch "" ""
<full-path>/linux32
<full-path>/linux64
hexdump -e ""
ping6 -
```

Evaluation: Cross-checking

KLEE can prove asserts on a per path basis

- Constraints have no approximations
- An assert is just a branch, and the constraint solver states feasibility/infeasibility of each branch
- If KLEE determines infeasibility of the false branch, then it proves that no value exists on the current path that could violate the assertion

Evaluation: Crosschecking

Assume f(x) and f'(x) implement the same interface:

1. Make input x symbolic
2. Run KLEE on assert(f(x) == f'(x))
3. For each path:
    a. *Terminate w/o error: paths are equivalent*
    b. *Terminate w/ error: mismatch found*

```
unsigned mod_opt(unsigned x, unsigned y) {
  if((y & −y) == y) // power of two?
    return x & (y−1);
  else
    return x % y;
}
unsigned mod(unsigned x, unsigned y) {
  return x % y;
}
int main() {
   unsigned x,y;
   make_symbolic(&x, sizeof(x));
   make_symbolic(&y, sizeof(y));
   assert(mod(x,y) == mod_opt(x,y));
   return 0;
}
```

Evaluation: Crosschecking

Input	BUSYBOX	COREUTILS
`comm t1.txt t2.txt`	[does not show difference]	[shows difference]
`tee -`	[does not copy twice to stdout]	[does]
`tee "" <t1.txt`	[infinite loop]	[terminates]
`cksum /`	`"4294967295 0 /"`	`"/: Is a directory"`
`split /`	`"/: Is a directory"`	
`tr`	[duplicates input on stdout]	`"missing operand"`
`[0 ``<'' 1]`		`"binary operator expected"`
`sum -s <t1.txt`	`"97 1 -"`	`"97 1"`
`tail -2l`	[rejects]	[accepts]
`unexpand -f`	[accepts]	[rejects]
`split -`	[rejects]	[accepts]
`ls --color-blah`	[accepts]	[rejects]
t1.txt: a *t2.txt:* b		

Discussions

- Strengths / Weaknesses
- Other solutions to handle environment interactions?

Discussions

- Other solutions to handle environment interactions?
  - *Executing calls to the environment directly*
  - *Modeling the environment*
  - *Forking the entire system state*

Thanks!

Q & A

References

- https://www.doc.ic.ac.uk/~cristic/talks/klee-stanford-2009.ppsx
- https://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec14-SymExecPapers.pdf
- https://www.cs.umd.edu/~mwh/se-tutorial/symbolic-exec.pdf