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

Cadar, Dunbar, Engler — OSDI 2008

Agenda

1. **Motivation**: Why is testing systems code hard?
2. **Background**: Symbolic execution 101
3. **KLEE Overview**: What is KLEE?
4. **KLEE Internals**: Capabilities, optimizations, scheduling
5. **Evaluation**: GNU Coreutils, Busybox, and beyond
6. **Demo**
7. **Strengths, Weaknesses & Review**

Motivation

The Problem

Testing real-world systems code is fundamentally hard:

Random Testing (aka Blackbox Fuzzing)

Pros: Fast, Scalable.

Cons: Blind, Low coverage

Symbolic Execution (KLEE is here)

Pros: Systematic, High coverage

Cons: Scalability, Solver dependancy

Formal Verification

Pros: Complete, Sound, Undisputed Method

Cons: **Complete** lack of scalability

What We Want (And KLEE Does)

An automated tool that can:

1. Generate **high-coverage** test inputs
2. Find **real bugs** (No False Positives)
3. Model the **environment** — filesystem, args, stdin

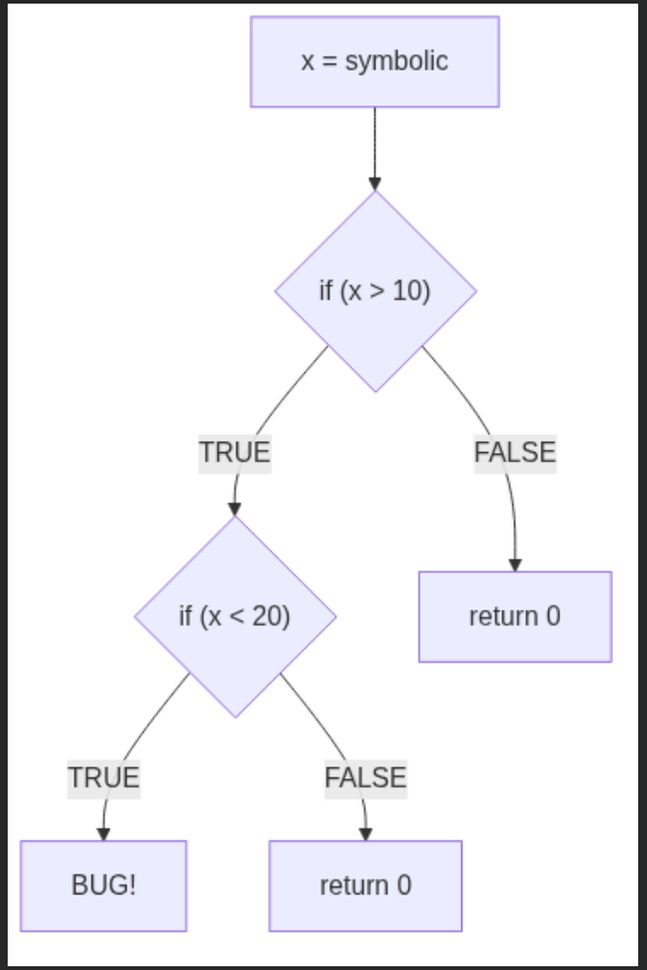
Symbolic Execution 101

Instead of running a program with **concrete** values, run it with **symbolic** variables that represent *all possible inputs*.

```
int foo(int x) {
    if (x > 10)        // Branch 1
        if (x < 20)    // Branch 2
            bug();
    return 0;
}
```

With concrete input `x = 5`: takes one path, misses the bug.

With **symbolic** `x`: explores *both* sides of every branch.



Path Conditions and Constraint Solving

Each explored path accumulates a **path condition** (PC):

```
Path to bug():  PC = (x > 10) ^ (x < 20)
```

When a path terminates or hits a target, we ask an **SMT solver**:

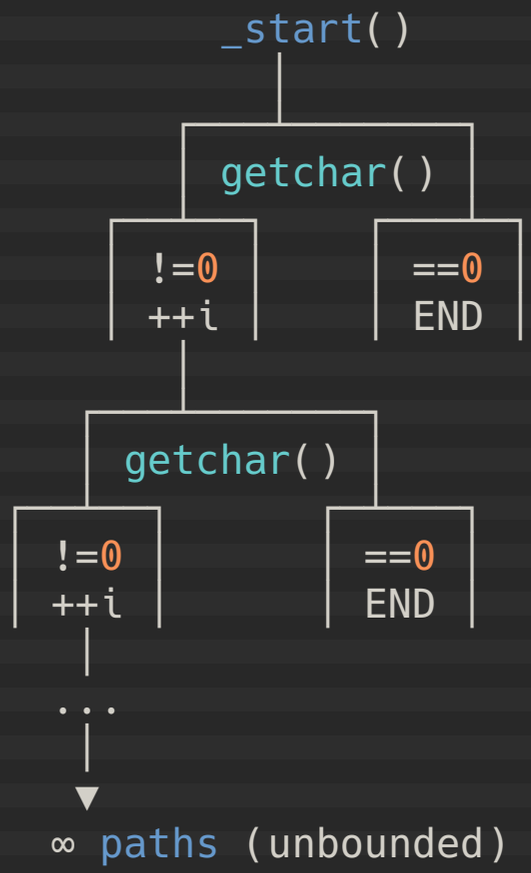
> "Give me a concrete value of x satisfying this PC"

Solver returns: `x = 15` → this is your **test case**.

Key insight: Each satisfying assignment is a concrete test input that exercises exactly that program path.

The Fundamental Challenge: Path Explosion

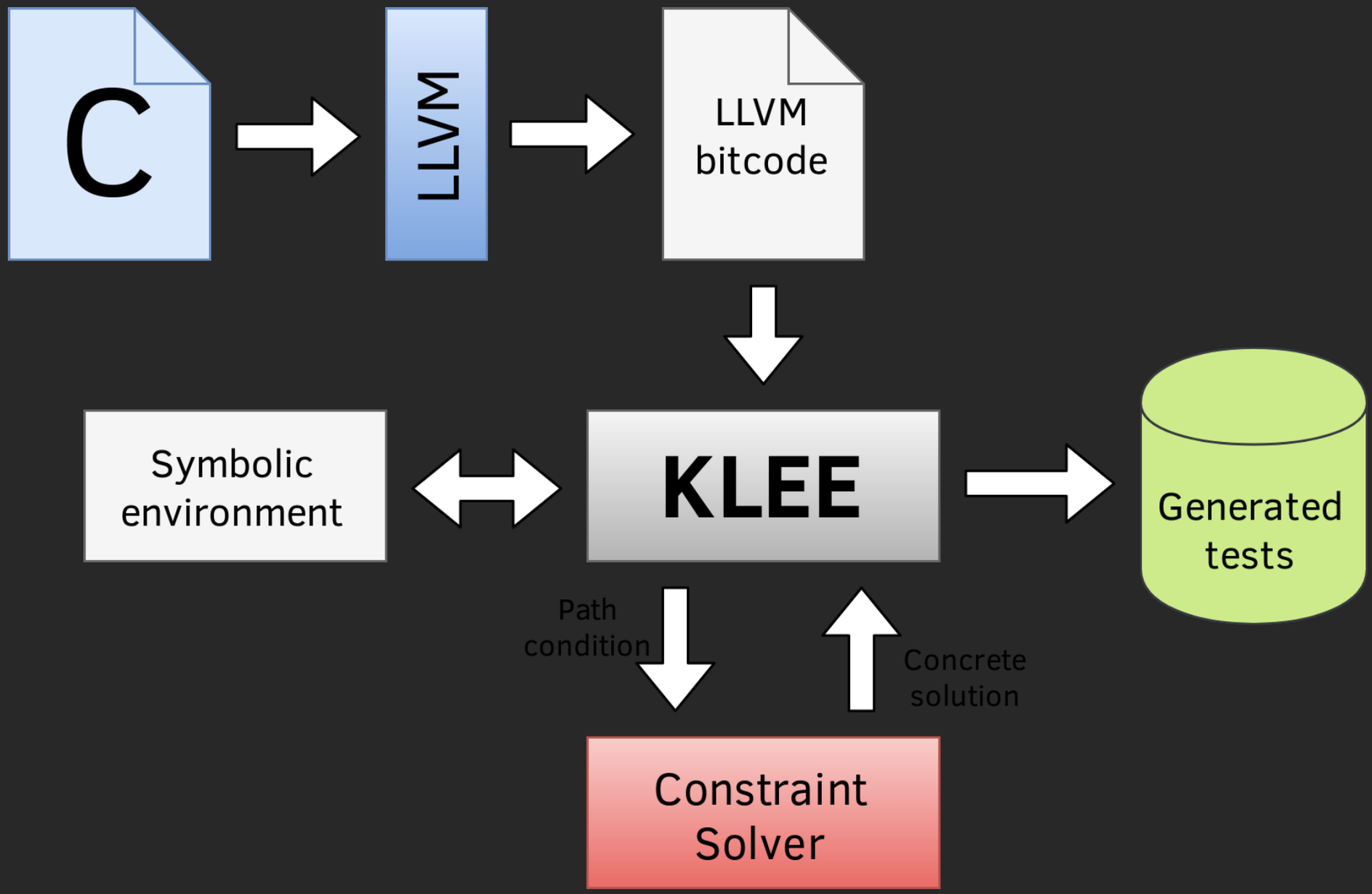
```
while(getchar())
    ++i;
```



KLEE's contribution is largely about *engineering around* path explosion, at scale, not solving it theoretically.

KLEE Architecture

The Full Pipeline



At each branch on a symbolic condition:

1. Query solver: can the condition be **true**? Can it be **false**?
2. If both → **fork** the state (duplicate everything)
3. Each child gets an updated path condition
4. Continue exploring both independently

Execution States and Forking

- Execution State:
  - Path Condition
  - Stack
  - Heap
  - PC/Register File

Object-level copy-on-write: Only modified memory objects are copied at fork time. The heap is an immutable map — structure is shared.

Finer than OS page-level COW — critical for **thousands of states**.

KLEE is an interpreter/OS for Execution States.

Environment Modeling

Programs usually read values from their environment:

- command-line arguments
- environment variables
- file data

KLEE handles the environment by redirecting calls to models.

Models are C code that intercept system calls, handling concrete and symbolic values.

KLEE uses those models to also test branches for syscall failure.

Example: Modeling read()

```
ssize_t read(int fd, void *buf, size_t count) {
    /* is fd concrete? */
    if (fd_is_concrete(fd)) {
        return pread(real_fd, buf, count, offset);
    } else {
        /* copy desired bytes from fixed-size sym file */
        for (i = 0; i < count; i++)
            buf[i] = sym_file->contents[offset + i];
    }
}
```

The pread() call here allows for many states to effeciently access a single fd.

Key Innovations

Search Heuristics

How do you pick which state to explore next?

KLEE interleaves **two strategies** in round-robin:

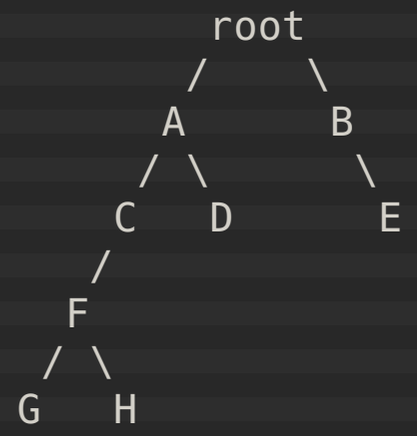
1. Random Path Selection

- Maintains the full execution tree
- Walks from root, randomly choosing left/right at each fork
- **Effect**: shallow, unexplored states get higher probability
- Prevents deep paths from starving exploration

2. Coverage-Optimized Search

- Prioritizes states most likely to increase coverage
- Weights each state by:
  - Distance to nearest uncovered instruction
  - Whether it recently covered new code
  - The callstack of the state

Random Path Selection — Visualized



Naïve round-robin: G, H, D, E each get 1/4 chance
→ Deep subtree under A dominates exploration

Random walk from root:
  root → A (50%) → C (25%) → F (12.5%) → G or H (6.25% each)
  root → A (50%) → D (25%)
  root → B (50%) → E (50%)

→ B/E gets 50% despite being only 1 state!
→ Prevents starvation of shallow, unexplored branches

Constraint Solving — The Bottleneck

The solver is where KLEE spends most of its time (without optimizations).

1. Expression Simplification

Rewrite and fold constants before sending anything to the solver. (e.g. strength reduction, linear simplification)

2. Constraint Set Simplification

3. Implied Value Concretization

Constraint Solving — The Bottleneck

The solver is where KLEE spends most of its time (without optimizations).

4. Constraint Independence

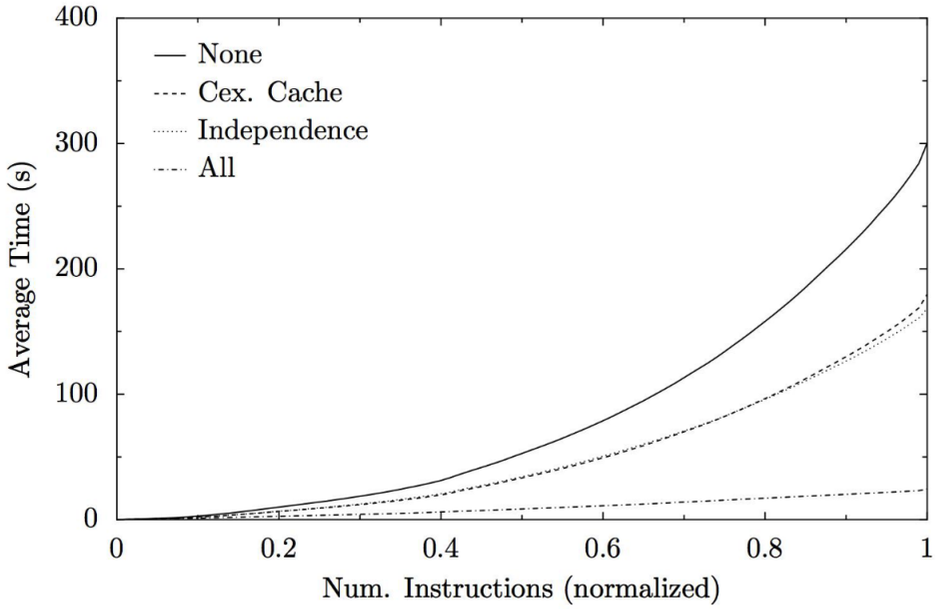
For example, given the constraint set `{i < j, j < 20, k > 0}`, a query of whether `i = 20` just requires the first two constraints.

5. Counterexample Caching

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*

Solver Optimization Impact

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



Someone is unenthusiastic about your work.

Evaluation

Flagship Result: GNU Coreutils

Tested **all 89** standalone Coreutils programs.

Coverage:

- Average: **>90% line coverage**
- Median: **>94%**
- Beat the developers' own hand-written test suite
- Most tools: 2K-12K ELOC (including library code)

Bugs Found:

- **56 serious bugs** across 452 applications (430K+ LoC)
- 3 bugs in Coreutils **missed for 15+ years**
- Bugs include memory errors, crashes, assertion failures
- All with concrete inputs

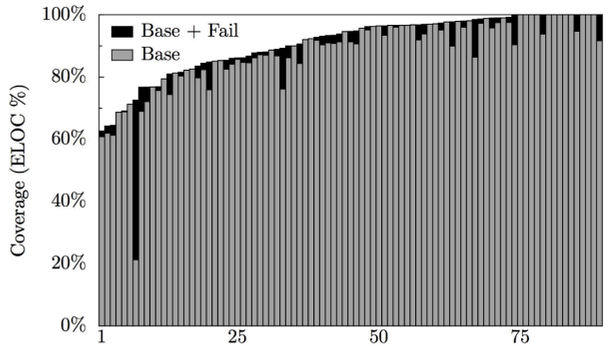


Figure 5: Line coverage for each application with and without failing system calls.

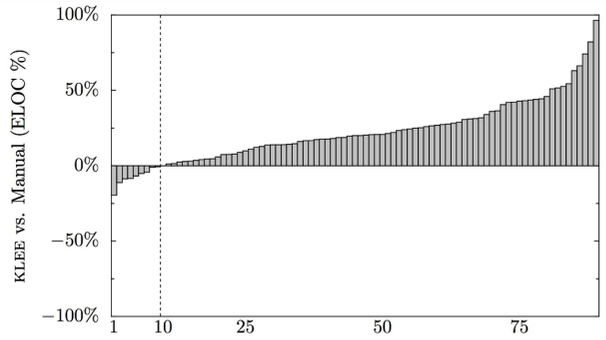


Figure 6: Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests (L_{man}) from KLEE tests (L_{klee}) and dividing by the total possible: $(L_{klee} - L_{man})/L_{total}$. Higher bars are better for KLEE,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

coreutils vs Busybox

Busybox results:

- Tested 75 equivalent utilities
- Even better coverage than Coreutils
- **100% coverage on 31 of 75** tools
- Found additional bugs in the embedded-system implementations

Cross-checking:

- Ran equivalent Coreutils & Busybox tools on the **same symbolic inputs**
- Compared outputs
- Found:
  - Serious correctness errors
  - Missing functionality
  - Behavioral inconsistencies

This demonstrates a powerful use beyond bug-finding: **differential testing via symbolic execution**.

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		

Table 3: Very small subset of the mismatches KLEE found between the BUSYBOX and COREUTILS versions of equivalent utilities. The first three are serious correctness errors; most of the others are revealing missing functionality.

Coverage: KLEE vs. Developers

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%

Table 2: Number of COREUTILS tools which achieve line coverage in the given ranges for KLEE and developers' tests (library code not included). The last rows shows the aggregate coverage achieved by each method and the average and median coverage per application.

KLEE **exceeded** the coverage of the developers' own test suite on most utilities — the most heavily tested open-source suite.

Demo Time

Strengths, Weaknesses and Review

Strengths

1. **Real-world evaluation**
2. **No human guidance**
3. **Concrete witnesses** — Every bug comes with a reproducing input; zero false positives
4. **Engineering depth** — COW states, solver caching, search heuristics
5. **Open-sourced**
6. **Modular architecture** — Search heuristics, solver stages, and environment models are all pluggable and extensible

Weaknesses

1. **Path explosion is unsolved** — Heuristics help, but deep loops and complex branching still defeat KLEE
2. **Environment models are manual** — ~2500 LoC of hand-written POSIX models; extending to new APIs is labor-intensive
3. **Requires source code**
4. **Constraint Solving is NP-hard (at best)**

Review

KLEE is the first work of its kind that demonstrates symbolic execution as a feasible real world program testing technique. It's simple, platform agnostic, explicitly searches for new paths AND WORKS!!! Unlike many other, even modern, works has no false positives and provides a clear method to reproduce its findings. It's a work of serious engineering, with multiple techniques bleeding edge for its time, (COW, cache, heuristics).

While it has optimizations for the contraints it hasn't been able to overcome the path explosion problem, unlike other works. Although it is presented to work well in the evaluation section, the programs used are the perfect paradigm for such testing, and fails to scale well in larger systems.

ALmost 20 years later, KLEE's open problems (environment modeling, oracle quality) **remain open**

Subsequent Work

S2E: Selective Symbolic Execution. Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, George Candea. 5th Workshop on Hot Topics in System Dependability (HotDep), Lisbon, Portugal, June 2009

Merge-Point: Enhancing symbolic execution with veritesting. Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, David Brumley. ICSE 2014: Proceedings of the 36th International Conference on Software Engineering

References

Paper: Cadar, Dunbar, Engler. *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*. OSDI 2008.

Code: github.com/klee/klee

Survey: Cadar, Sen. *Symbolic Execution for Software Testing: Three Decades Later*. CACM 2013.

Slides: Liu, Sun, Hu. *UMich EECS 583 KLEE Presentation*. 2018.

Thank you — Questions?