Baskin Engineering
UC SANTA CRUZ

KLEE:

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

Cristian Cadar, Daniel Dunbar, Dawson Engler
Stanford University

Presented by: Matthew Rhea

Overview of KLEE

Baskin Engineering
UC SANTA CRUZ

- A new symbolic execution tool that supersedes EXE

- Leverages lessons from EXE to check a broader range of complex systems applications

  Query optimization and search heuristics are drawn from EXE, but KLEE is a complete, expansive redesign

Overview of KLEE

Baskin Engineering
UC SANTA CRUZ

- Achieves high code coverage

  Employs a variety of search heuristics and constraint solving optimizations, and represents program states compactly in the re-designed OS-Interpreter hybrid architecture

- Generates more tests than handwritten, manual test suites in all experiments

Overview of KLEE

Baskin Engineering
UC SANTA CRUZ

- Applicable to real-world, complex programs

  KLEE, as opposed to EXE, deals with the external environment of the programs

  “Out of the box” checking of programs

- Can discover functional incorrectness

  KLEE is limited to low-level programming errors, ***but*** can leverage its single-bit accuracy of its constraints to verify functional correctness

Overview of KLEE

Baskin Engineering
UC SANTA CRUZ

- There are more optimizations than EXE

  New optimizations are better implemented and allow for greater test coverage as a result

- Capable of executing a large number of states simultaneously

The Architecture of KLEE

Baskin Engineering
UC SANTA CRUZ

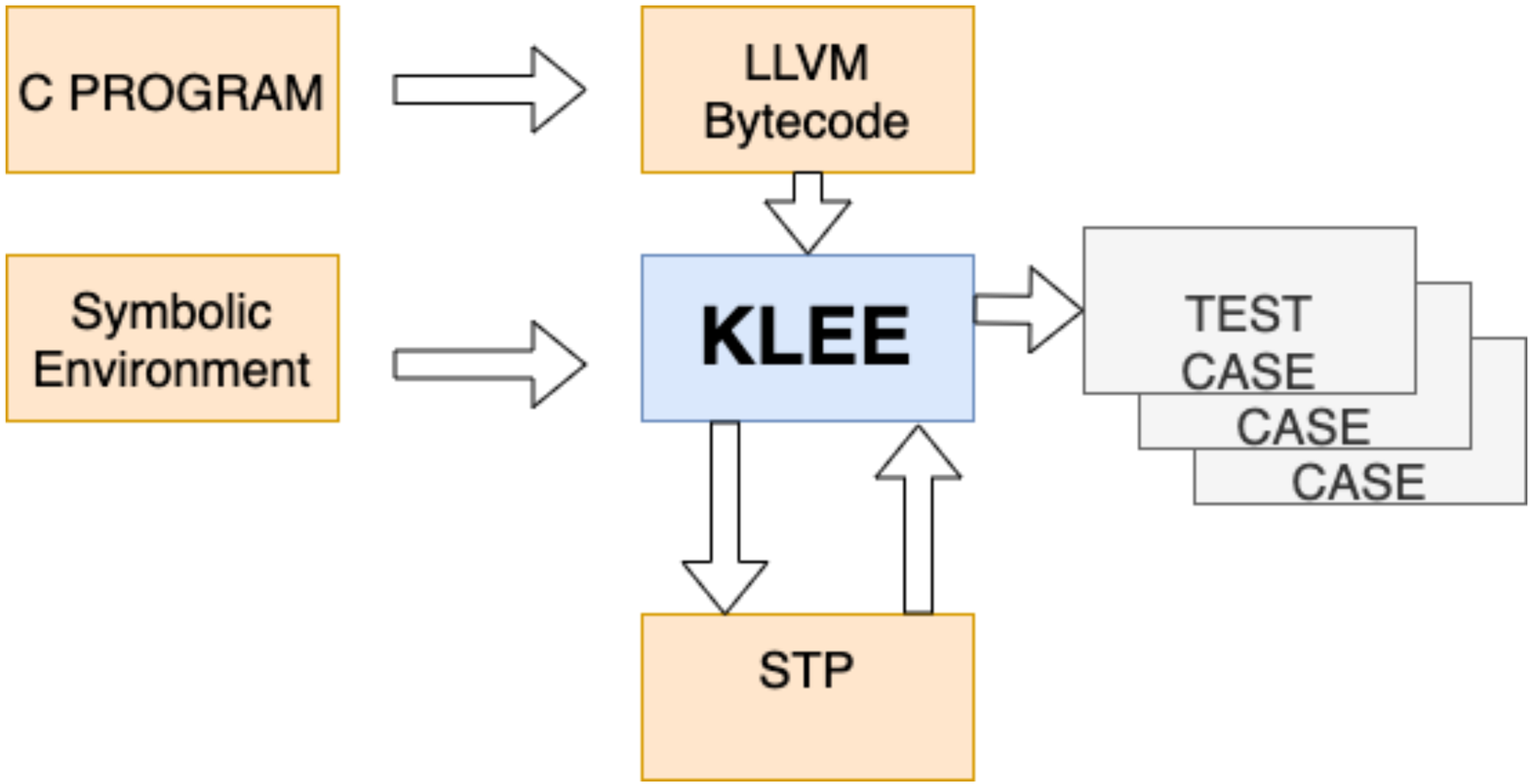
- A hybrid between an ***operating system*** for symbolic process and an ***interpreter***
  Symbolic processes are similar to normal machine processes in that each has a register file, stack, heap, PC, and path condition (otherwise called a *state*)

- Compiles programs to the LLVM assembly language
  KLEE interprets this instruction set and maps to constraints

- Uses the constraint solver STP for deciding branching conditions
  When program execution branches based on a symbolic value, KLEE maintains a set of constraints that correspond to the *path condition*

- Handles program environment with symbolic models

The Architecture of KLEE[1]



[1]: Diagram structure inspired by the diagram from: Yoni Leibowitz

The Architecture of KLEE: OS for Symbolic Processes

Baskin Engineering
UC SANTA CRUZ

- The OS for symbolic processes represents a huge leap from how EXE handled symbolic execution
  Unlike normal machine processes, storage locations for a state refer to expressions (formatted as a tree), instead of raw data values

  Interior nodes of the tree are LLVM assembly operations, while leaves are symbolic variables or constants (in the case of locations for constants we call these *concrete*)

- Conditional branches use boolean expressions (i.e. a *branch condition*) to decide which branch to take

The Architecture of KLEE: OS for Symbolic Processes

Baskin Engineering
UC SANTA CRUZ

How to symbolically execute LLVM instructions?

```
%dst = add i32 %src0, %src1
```

The Architecture of KLEE: OS for Symbolic Processes

Baskin Engineering
UC SANTA CRUZ

How to symbolically execute LLVM instructions?

%dst = add i32 %src0, %src1

KLEE retrieves and addends from %src0 and %src1 registers and writes a new expression Add(%src0, %src1) to the %dst register. Code that builds expressions checks if all given operands are concrete, and, if so, performs the operation natively, returning a constant expression.

The Architecture of KLEE: OS for Symbolic Processes

- Number of states grow quickly
  EXE manages this by using one native operating system process per state

- Compared to EXE, KLEE drastically reduces memory requirements for states by internalizing state representation in its own custom symbolic processes

The Architecture of KLEE: Core Interpreter Loop

- At the core of KLEE is an interpreter of the current state (symbolic process)

  The interpreter loop selects a state to run and then performs symbolic execution of a single instruction in the context of that state

  This loop continues until no states remain (or a user-defined timeout is reached)

- KLEE is capable of executing multiple states simultaneously

Usage: MINIX *tr*

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                             9
3 :   while (*arg) {                                         10*
4 :     if (*arg == '\\') {                                  11*
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:          i++;
11:        } while (i<4 && *arg>='0' && *arg<='7');
12:        *buffer++ = ac;
13:      } else if (*arg != '\0')
14:        *buffer++ = *arg++;
15:    } else if (*arg == '[') {                             12*
16:      arg++;                                              13
17:      i = *arg++;                                         14
18:      if (*arg++ != '-') {                                15!
19:        *buffer++ = '[';
20:        arg -= 2;
21:        continue;
22:      }
23:      ac = *arg++;
24:      while (i <= ac) *buffer++ = i++;
25:      arg++;       /* Skip ']' */
26:    } else
27:      *buffer++ = *arg++;
28:  }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                         2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                  4
35:   }                                                      5
36:   ...                                                    6
37:   expand(argv[index++], index);                          7
38:   ...
39: }
```

What would cause KLEE to branch?

Path conditions such as “argc > 1” at line 33, and other conditional checks in the code (*)

Dangerous operations?

Dereferencing pointers and assertions. These generate branches that KLEE must check if values exist that cause an error

Usage: MINIX *tr*

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                             9
3 :   while (*arg) {                                         10*
4 :     if (*arg == '\\') {                                  11*
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:          i++;
11:        } while (i<4 && *arg>='0' && *arg<='7');
12:        *buffer++ = ac;
13:      } else if (*arg != '\0')
14:        *buffer++ = *arg++;
15:    } else if (*arg == '[') {                             12*
16:      arg++;                                              13
17:      i = *arg++;                                         14
18:      if (*arg++ != '-') {                                15!
19:        *buffer++ = '[';
20:        arg -= 2;
21:        continue;
22:      }
23:      ac = *arg++;
24:      while (i <= ac) *buffer++ = i++;
25:      arg++;       /* Skip ']' */
26:    } else
27:      *buffer++ = *arg++;
28:  }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                         2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                  4
35:   }                                                      5
36:   ...                                                    6
37:   expand(argv[index++], index);                          7
38:   ...
39: }
```

KLEE has two goals here:

1) Hit every line of executable code in the program

2) Detect at each dangerous operation if any input value exists that would cause an error

Usage: MINIX *tr*

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                             9
3 :   while (*arg) {                                         10*
4 :     if (*arg == '\\') {                                  11*
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:          i++;
11:        } while (i<4 && *arg>='0' && *arg<='7');
12:        *buffer++ = ac;
13:      } else if (*arg != '\0')
14:        *buffer++ = *arg++;
15:    } else if (*arg == '[') {                             12*
16:      arg++;                                              13
17:      i = *arg++;                                         14
18:      if (*arg++ != '-') {                                15!
19:        *buffer++ = '[';
20:        arg -= 2;
21:        continue;
22:      }
23:      ac = *arg++;
24:      while (i <= ac) *buffer++ = i++;
25:      arg++;       /* Skip ']' */
26:    } else
27:      *buffer++ = *arg++;
28:  }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                         2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                  4
35:   }                                                      5
36:   ...                                                    6
37:   expand(argv[index++], index);                          7
38:   ...
39: }
```

Assuming KLEE has these set options:

1) Constrain number of input arguments to be between 0 and 3

2) Size of each argument is 1, 10, and 10 characters respectively

Usage: MINIX *tr*

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                             9
3 :   while (*arg) {                                         10*
4 :     if (*arg == '\\') {                                  11*
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:          i++;
11:        } while (i<4 && *arg>='0' && *arg<='7');
12:        *buffer++ = ac;
13:      } else if (*arg != '\0')
14:        *buffer++ = *arg++;
15:    } else if (*arg == '[') {                             12*
16:      arg++;                                              13
17:      i = *arg++;                                         14
18:      if (*arg++ != '-') {                                15!
19:        *buffer++ = '[';
20:        arg -= 2;
21:        continue;
22:      }
23:      ac = *arg++;
24:      while (i <= ac) *buffer++ = i++;
25:      arg++;       /* Skip ']' */
26:    } else
27:      *buffer++ = *arg++;
28:  }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                         2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                  4
35:   }                                                      5
36:   ...                                                    6
37:   expand(argv[index++], index);                          7
38:   ...
39: }
```

Execution starts…

1. Branch at *argc > 1* at line 33
2. Invoke STP on current path condition to decide that both directions are possible.
3. Both are possible, so fork execution and follow both paths
4. KLEE must pick one to execute to first following a state scheduling algorithm
5. At each dangerous operation along the path, KLEE checks for any possible value

Usage: MINIX *tr*

```
1 : void expand(char *arg, unsigned char *buffer) {          8
2 :   int i, ac;                                             9
3 :   while (*arg) {                                         10*
4 :     if (*arg == '\\') {                                  11*
5 :       arg++;
6 :       i = ac = 0;
7 :       if (*arg >= '0' && *arg <= '7') {
8 :         do {
9 :           ac = (ac << 3) + *arg++ - '0';
10:          i++;
11:        } while (i<4 && *arg>='0' && *arg<='7');
12:        *buffer++ = ac;
13:      } else if (*arg != '\0')
14:        *buffer++ = *arg++;
15:    } else if (*arg == '[') {                             12*
16:      arg++;                                              13
17:      i = *arg++;                                         14
18:      if (*arg++ != '-') {                                15!
19:        *buffer++ = '[';
20:        arg -= 2;
21:        continue;
22:      }
23:      ac = *arg++;
24:      while (i <= ac) *buffer++ = i++;
25:      arg++;        /* Skip ']' */
26:    } else
27:      *buffer++ = *arg++;
28:  }
29: }
30: ...
31: int main(int argc, char* argv[]) {                        1
32:   int index = 1;                                         2
33:   if (argc > 1 && argv[index][0] == '-') {               3*
34:     ...                                                  4
35:   }                                                      5
36:   ...                                                    6
37:   expand(argv[index++], index);                          7
38:   ...
39: }
```

Execution continues…

1. At line 18, KLEE determines input exists that allow *arg* to go out of bounds: at line 15 we increment *arg* without checking if the string has ended
2. If it has, this increment skips the terminating ‘\0’ and points to invalid memory

KLEE generates the concrete value: *tr [“” “”*

Environmental Modeling

Baskin Engineering
UC SANTA CRUZ

- Code reads values from its environment
  Environmental variables, command-line arguments, file data and metadata, network packets

- In order to generate constraints, redirect calls that access *models of the environment*
  C code models of ~40 system calls are implemented to handle interactions with environmental dependencies

- Private to each state is a symbolic file system
  User defines size of files and number of symbolic files (N) in the symbolic file system

- Applications may want to evaluate failing system calls
  KLEE can simulate environmental failures by failing system calls in a controlled manner
  For example, system calls such as *write()* fail because of a full disk

Environmental Modeling: *read()* KLEE model

```
1 : ssize_t read(int fd, void *buf, size_t count) {
2 :   if (is_invalid(fd)) {
3 :     errno = EBADF;
4 :     return -1;
5 :   }
6 :   struct klee_fd *f = &fds[fd];
7 :   if (is_concrete_file(f)) {
8 :     int r = pread(f->real_fd, buf, count, f->off);
9 :     if (r != -1)
10:       f->off += r;
11:     return r;
12:   } else {
13:     /* sym files are fixed size: don't read beyond the end. */
14:     if (f->off >= f->size)
15:       return 0;
16:     count = min(count, f->size - f->off);
17:     memcpy(buf, f->file_data + f->off, count);
18:     f->off += count;
19:     return count;
20:   }
21: }
```

File system operations check if the action is from a concrete file on disk or a symbolic file

If symbolic, invoke the emulation of the operations effect on a simple symbolic file system, private to each state

Environmental Modeling: *read()* KLEE model

```
1 : ssize_t read(int fd, void *buf, size_t count) {
2 :   if (is_invalid(fd)) {
3 :     errno = EBADF;
4 :     return -1;
5 :   }
6 :   struct klee_fd *f = &fds[fd];
7 :   if (is_concrete_file(f)) {
8 :     int r = pread(f->real_fd, buf, count, f->off);
9 :     if (r != -1)
10:       f->off += r;
11:     return r;
12:   } else {
13:     /* sym files are fixed size: don't read beyond the end. */
14:     if (f->off >= f->size)
15:       return 0;
16:     count = min(count, f->size - f->off);
17:     memcpy(buf, f->file_data + f->off, count);
18:     f->off += count;
19:     return count;
20:   }
21: }
```

A concrete call:

int fd = open("/etc/fstab", O_RDNLY);

A symbolic call:

int fd = open(argv[1], O_RDNLY);

Environmental Modeling: *read()* KLEE model

```
1 : ssize_t read(int fd, void *buf, size_t count) {
2 :   if (is_invalid(fd)) {
3 :     errno = EBADF;
4 :     return -1;
5 :   }
6 :   struct klee_fd *f = &fds[fd];
7 :   if (is_concrete_file(f)) {
8 :     int r = pread(f->real_fd, buf, count, f->off);
9 :     if (r != -1)
10:       f->off += r;
11:     return r;
12:   } else {
13:     /* sym files are fixed size: don't read beyond the end. */
14:     if (f->off >= f->size)
15:       return 0;
16:     count = min(count, f->size - f->off);
17:     memcpy(buf, f->file_data + f->off, count);
18:     f->off += count;
19:     return count;
20:   }
21: }
```

A concrete call:

Use operating system *pread()* to read contents (7 - 11)

A symbolic call:

read() copies from the underlying symbolic buffer holding the file contents into the user supplied buffer (13 - 19)

Compact State Representation Optimization

Baskin Engineering
UC SANTA CRUZ

- Compact State Representation
  Refers to the process-per-symbolic-state representation KLEE has (as opposed to EXE's native OS process per state)

  Implements copy-on-write at object level to dramatically reduce per-state memory Requirements

  Heap is immutable map, so portions of the heap structure itself can be shared amongst multiple states (in constant time)

Constraint Solving Optimizations: Expression Rewriting

Baskin Engineering
UC SANTA CRUZ

- Simple compiler-like optimizations such as:

(x * 0 = 0) [arithmetic simplifications]

(x * 2^n = x << n) [strength reduction]

(2*x - x = x) [linear simplification]

Constraint Solving Optimizations: Constraint Set Simplification

- Symbolic execution typically involves large number of additions of constraints to path condition

- KLEE rewrites constraints as new equality constraints are added to the constraint set

$$(x < 10) \text{ AND } (x = 5) \Rightarrow \text{TRUE}$$

Constraint Solving Optimizations: Implied Value Concretization

- KLEE can determine when variables in constraints become implicitly concrete

$$x + 1 = 10$$

Constraint Solving Optimizations: Constraint Independence

- Taken directly from EXE!
- Divides constraint sets into disjoint independent subsets back on the symbolic variables they reference
- Eliminate irrelevant constraints prior to querying STP

$$\{i < j, j < 20, k > 0\} \text{ [Query: } i = 20\text{]}$$

Constraint Solving Optimizations: Counter-Example Cache

Baskin Engineering
UC SANTA CRUZ

- Simplifies redundant querying of STP
- Maps sets of constraints to counter-examples
- Allows efficient searching for cache entries for both supersets and subsets of constraint sets

- Three ways to eliminate queries:
  1. If subset has no solution, constraint set has no solution
  2. If superset has a solution, that solution also satisfies original constraint set
  3. If subset has a solution, it is “likely” that this is a solution of the original constraint set

Search Heuristics for State Scheduling (More Optimizations!)

Baskin Engineering
UC SANTA CRUZ

- KLEE interleaves two search heuristics in a round-robin fashion
  1. Random Path Selection
  2. Coverage-Optimized Search

- Random Path Selection

  Binary tree recording of program path in which internal nodes are forked executions and leaves are current states

  Randomly traverse from root to ensure equal probability of choosing subtrees

- Coverage-Optimized Search

  Compute weight for each state (according to “heuristics”) and randomly select a state according to these weights

Evaluation: Optimizations

Baskin Engineering
UC SANTA CRUZ

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

Table 1: Performance comparison of KLEE's solver optimizations on COREUTILS. Each tool is run for 5 minutes without optimization, and rerun on the same workload with the given optimizations. The results are averaged across all applications.

Evaluation: Optimizations

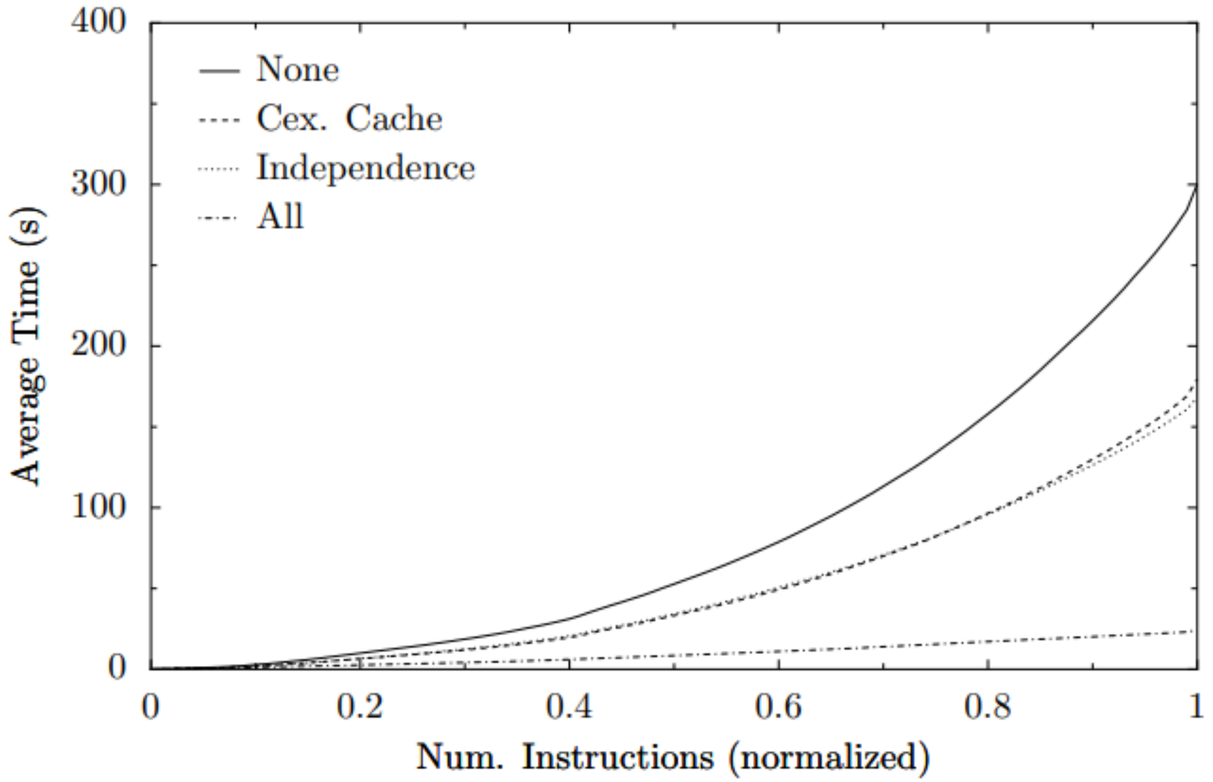


Figure 2: The effect of KLEE's solver optimizations over time, showing they become more effective over time, as the caches fill and queries become more complicated. The number of executed instructions is normalized so that data can be aggregated across all applications.

Evaluation: GNU COREUTILS Line Coverage

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.

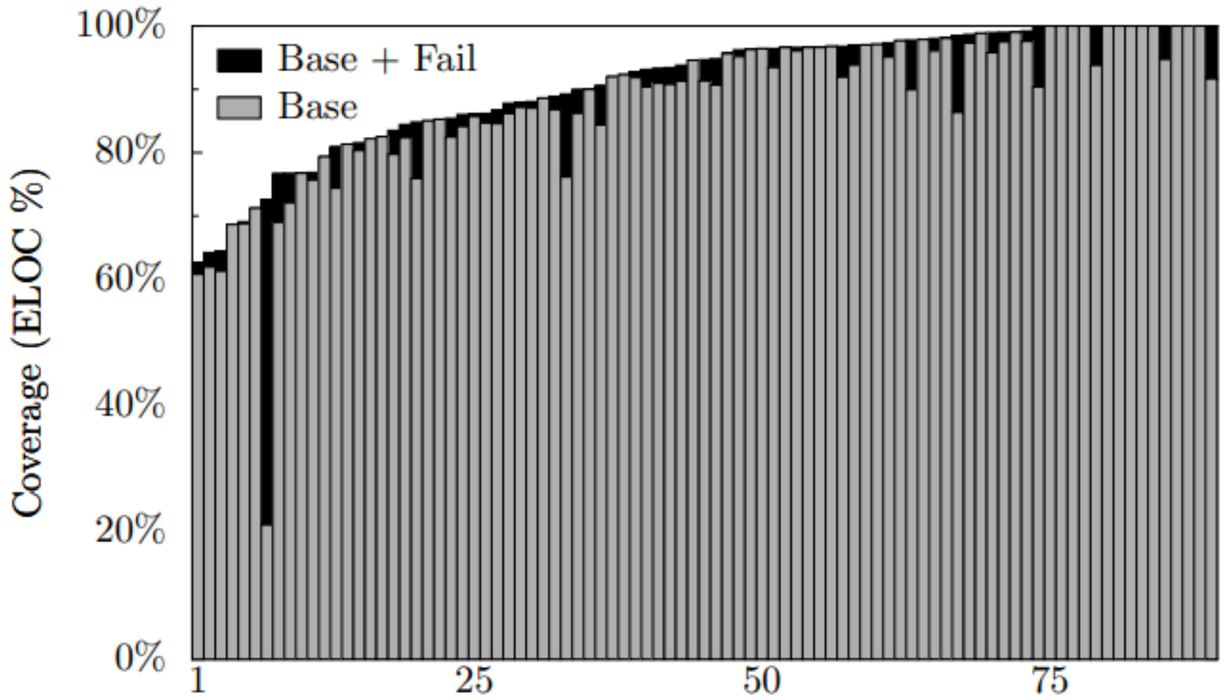


Figure 5: Line coverage for each application with and without failing system calls.

Evaluation: GNU COREUTILS Line Coverage

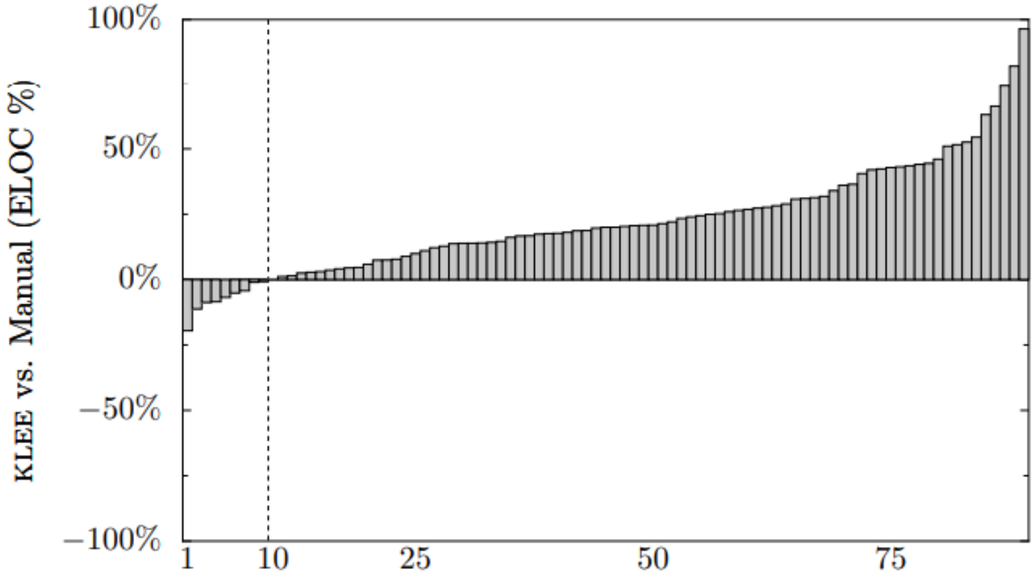


Figure 6: Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests (L_{man}) from KLEE tests (L_{klee}) and dividing by the total possible: $(L_{klee} - L_{man})/L_{total}$. Higher bars are better for KLEE,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

KLEE's generated tests beat developer suite in almost all 89 COREUTIL utilities

Limitation: KLEE does not validate that application output matches the expected output

Evaluation: GNU COREUTILS Bug-Finding

```
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\t"
t3.txt: "\n"
t4.txt: "a"

Figure 7: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.

10 unique bugs found

3 of which have existed since 1992

Evaluation: BUSYBOX Utilities Line Coverage

Baskin Engineering
UC SANTA CRUZ

	COREUTILS		BUSYBOX	
Coverage (w/o lib)	**KLEE tests**	**Devel. tests**	**KLEE tests**	**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.

BUSYBOX is an implementation of UNIX utilities for embedded systems that where overlap exists tries to replicate COREUTILS functionality

KLEE beats developer test suite by a factor of two

Evaluation: BUSYBOX Utilities Bug-Finding

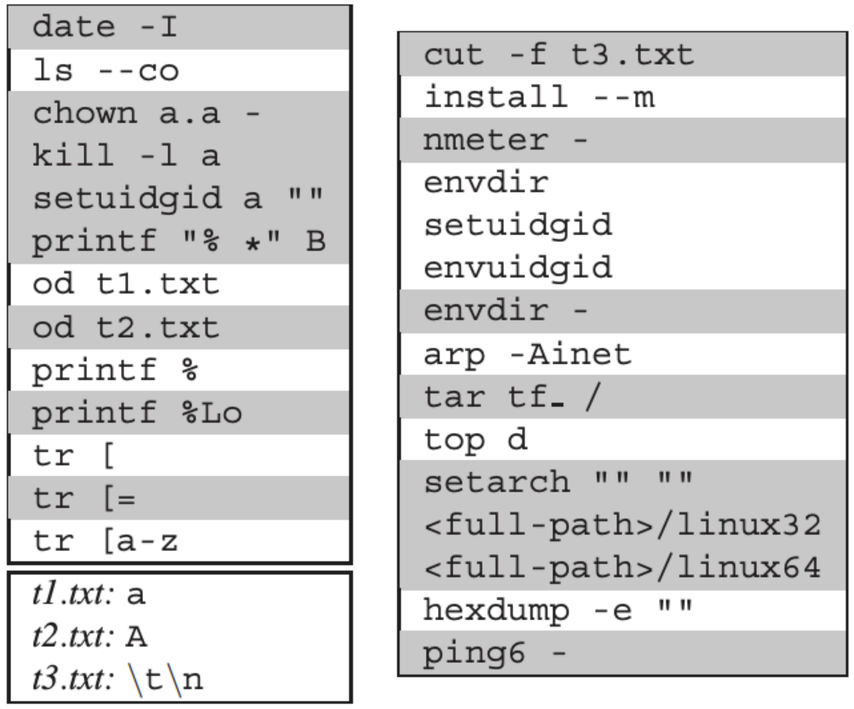


Figure 10: KLEE-generated command lines and inputs (modified for readability) that cause program crashes in BUSYBOX. When multiple applications crash because of the same shared (buggy) piece of code, we group them by shading.

KLEE tested 279 BUSYBOX tools and 84 MINIX tools

21 bugs found in both tools

Evaluation: HiStar OS Kernel Line Coverage

Baskin Engineering
UC SANTA CRUZ

An attempt by KLEE to check non-application code

Test	Random	KLEE	ELOC
With Disk	50.1%	67.1%	4617
No Disk	48.0%	76.4%	2662

Table 4: Coverage on the HiStar kernel for runs with up to three system calls, configured with and without a disk. For comparison we did the same runs using random inputs for one million trials.

Evaluation: HiStar OS Kernel Bug-Finding

Baskin Engineering
UC SANTA CRUZ

```
1 : uintptr_t safe_addptr(int *of, uint64_t a, uint64_t b) {
2 :     uintptr_t r = a + b;
3 :     if (r < a)
4 :         *of = 1;
5 :     return r;
6 : }
```

Figure 13: HiStar function containing an important security vulnerability. The function is supposed to set `*of` to true if the addition overflows but can fail to do so in the 32-bit version for very large values of `b`.

KLEE found a critical security bug in the 32-bit version of HiStar

safe_addptr sets **of* to true if addition overflows

The function can fail to indicate overflow for large values of b

Test should be: *(r < a || (r < b)*

Finding Deep Correctness Issues with Tool Equivalence

Baskin Engineering
UC SANTA CRUZ

- KLEE's constraints have perfect accuracy down to a single bit
  If at an *assert* the solver states the false branch of the *assert* cannot execute given current path conditions, then it has proved that no value exists on the current path that could violate the assert

- Deeper checks of code can be performed by leveraging this property of KLEE
  Consider two procedures implementing the same interface, *f* and *f'*

  If fed the same symbolic argument, functional equivalence can be tested on a per-path basis and asserting: *assert(f(x) == f'(x))*
- IF KLEE reaches an *assert* and finds no value that violates it, it has proven functional equivalence

Illustrating Equivalence Checking in KLEE

```
1 : unsigned mod_opt(unsigned x, unsigned y) {
2 :   if((y & −y) == y) // power of two?
3 :     return x & (y−1);
4 :   else
5 :     return x % y;
6 : }
7 : unsigned mod(unsigned x, unsigned y) {
8 :   return x % y;
9 : }
10: int main() {
11:     unsigned x,y;
12:     make_symbolic(&x, sizeof(x));
13:     make_symbolic(&y, sizeof(y));
14:     assert(mod(x,y) == mod_opt(x,y));
15:     return 0;
16: }
```

Figure 11: Trivial program illustrating equivalence checking. KLEE proves total equivalence when $y \neq 0$.

Comparing modulo implementations

Line 14 checks for differences in the symbolic inputs x and y

Two code paths reach this assert: The true path uses the solver to check the constraint holds for all values:

(y&-y) == y implies (x&(y-1)) == x%y

This query succeeds, and the false path checks the vacuous tautology that (y&-y) != y implies that x%y == x%y

KLEE proves equivalence for all non-zero values!

Equivalence Checking Results

Baskin Engineering
UC SANTA CRUZ

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.

KLEE finds mismatches when crosschecking BUSYBOX and COREUTIL implementations

The first three lines showcase hard correctness errors, others reveal missing functionality

Discussion

Klee is supposed to explore all paths, right? So, why did they not get 100% code coverage all the time?

Long-term goal is to auto-generate tests for any program. What are some bottlenecks here?