HARVARD

School of Engineering
and Applied Sciences

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

Cadar, Dunbar, and Engler

OSDI 08

CS252r Spring 2011

KLEE Overview

- Symbolic execution tool
  - Generates tests that achieve high coverage
    - Over 90% on average on ~160 user-level programs
  - Detect if dangerous operations could cause error
- Challenges:
  - Interacting with the environment
    - OS, network, user
  - Scaling to real-world code

Example: tr

```
llvm-gcc --emit-llvm -c tr.c -o tr.bc
```

```
klee --max-time 2 --sym-args 1 10 10
   --sym-files 2 2000 --max-fail 1 tr.bc
```

Example: 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: }
```

Constraint solver determines that execution `tr [ "" "" ` *will cause dereference of invalid memory*

Architecture

- Operates on LLVM bytecode
  - RISC-like virtual instruction set
  - Has a gcc frontend
- A **symbolic process** (or **state**) is the state of a symbolically executing process
  - Has register file, stack, heap, program counter, path condition
  - Storage locations (stack, heap, registers) contain **symbolic expressions**
    - Concrete values/constants, symbolic variables, arithmetic operations, bitwise manipulations, memory accesses, …
- Whenever symbolic execution encounters a branch, state is cloned, updating instruction pointer and path condition appropriately
  - If constraint solver determines path condition is false, state can be dropped
  - Potential errors are treated as branches
    - E.g., division generates a branch that checks for zero divisor

Scaling up

- Compact state representation
  - State cloned frequently
    - e.g., For Coreutils, average of about 50,000 states generated (with a cap on memory usage)
  - Use copy-on-write at object granularity
    - Allows much sharing of state
    - Constant-time state cloning
- Symbolic expressions over constants are simplified
  - e.g., Add(5, 3) is simplified to 8

Scaling up: query optimization

- Calling constraint solver dominates cost
  - So want to reduce/simplify calls
  - (is still about 40% of total time, even with optimizations)
- Expression rewriting
  - Strength reduction ($x * 2^n \mapsto x << n$)
    arithmetic simplification ($x + 0 \mapsto x$)
    linear simplification ($3*x + x \mapsto 4*x$)
- Constraint set simplification
  - Use equality constraints ($x=5$) to rewrite earlier constraints ($x < 10$) and simplify ($5 < 10 \mapsto$ true)
- Implied value concretization
  - e.g., $x+1=10$ implies $x=9$

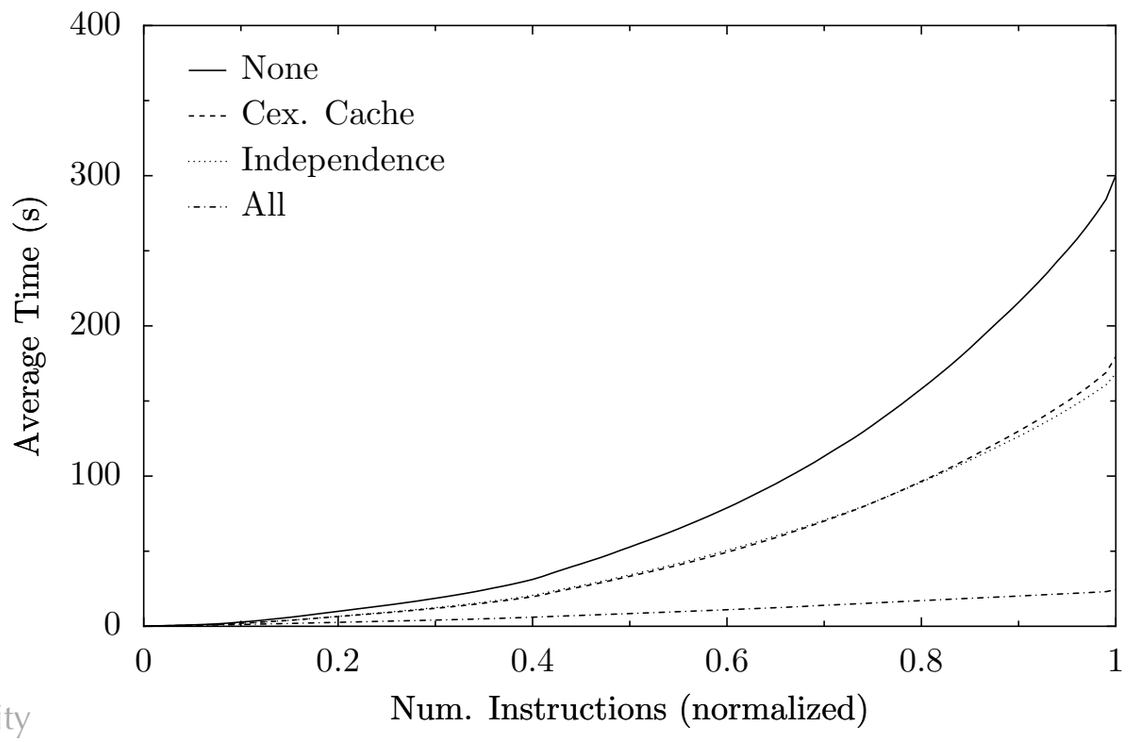
Scaling up: query optimization

- Constraint independence
  - Partition constraints based on which symbolic variables they access
  - Only gives relevant partitions to the solver
- Counter-example cache
  - Cache results of previous constraint solver queries
  - If constraint set C has no solution and $C \subseteq C'$, then neither does C'
    - Need to be able to efficiently search for subsets
  - If constraint set C has solution s and $C' \subseteq C$, then C' has solution s
    - Need to be able to efficiently search for supersets
  - If constraint set C has solution s and $C \subseteq C'$, then C' likely has solution s
    - Can cheaply check if solution s works for C'

Effect of cache

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.



State exploration

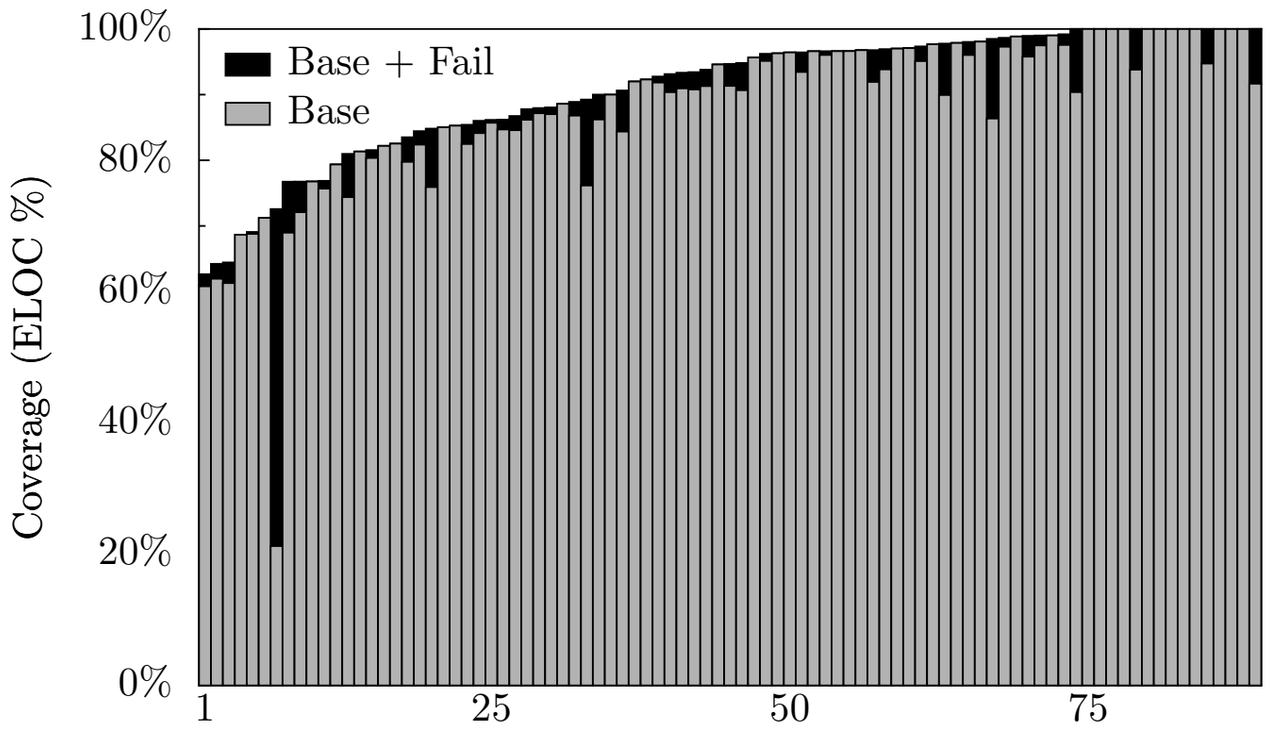
- Many concurrent states, representing different program executions
- Aim: get good coverage of code
- Problem: at each step, which state to choose to run?
- Answer: mix of two strategies
  - Random path selection
    - Maintain binary tree recording program path followed for all active states
    - Choose an active state by randomly traversing tree from root
    - Biased towards states higher in the tree; not biased in terms of number of active states
  - Coverage-Optimized Search
    - Compute weight for each state
      - Minimum distance to uncovered instruction, call stack of state, whether state recently covered new code
    - Randomly choose state according to weights

Environment

- Program interactive with environment
  - Command line args, environment variables, file data, network packets, …
- KLEE models semantics, and redirects library calls to these models
  - Models written in C, apparently without much coupling to KLEE internals
- E.g., Symbolic file system
  - Single directory with N symbolic files
  - Co-exists with real file system
    - If open called with concrete file name, will open real file
      - int fd = open("/etc/fstab", O_RDNLY);
    - If open called with symbolic file name, will form and match each of the N symbolic files (and also fail once)
      - int fd = open(argv[1], O_RDNLY);

Evaluation

- Results are reported in **line coverage**
  - Not the best metric. What would be better?
- Several evaluations: Coreutils, Busybox, HiStar
- Punchline: in reasonable time, got better (16.8%)coverage that tests manually developed over 15 years



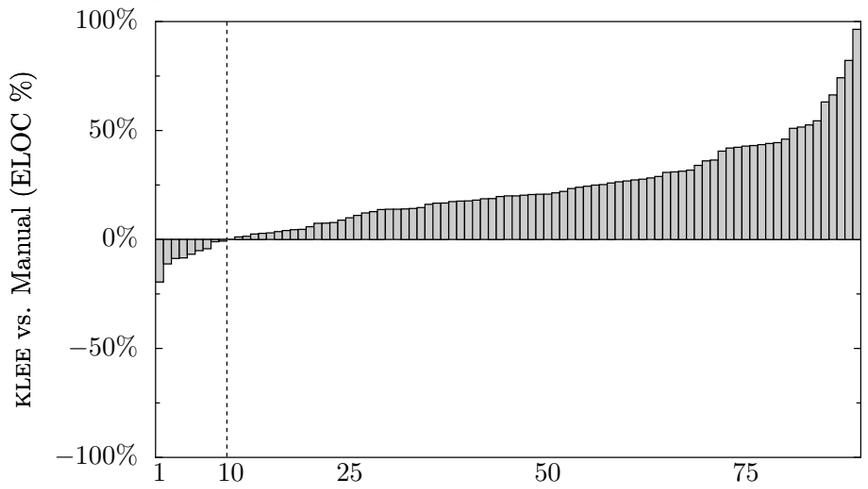


Figure 6: Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests (L_{man}) from KLEE tests (L_{klee}) and dividing by the total possible: $(L_{klee} - L_{man})/L_{total}$. Higher bars are better for KLEE,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

Bug finding

- Found ten unique bugs in Coreutils

```
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.

- Found 21 bugs in Busybox and 21 in Minix
  - all memory errors

Some additional points

- Produces test cases that can run outside of KLEE
  - Reduces impact of bugs in KLEE and non-determinism
- Symbolic execution for functional correctness?
  - Symbolic execution of two implementations of same interface
  - Find inputs on which functionality differs

HARVARD

School of Engineering
and Applied Sciences

Mixing Type Checking and Symbolic Execution

Phang, Chang, and Foster
PLDI 2010

CS252r Spring 2011

Mix Overview

- Type checking: imprecise but scalable
  - (Typically) Flow- and context-insensitive
- Symbolic execution: precise but inefficient
- Mix: combines symbolic execution and type checking
  - More precise than just type checking, more efficient than symbolic execution
  - Provably sound
    - And mostly re-uses proofs for type-soundness and sound symbolic execution

Motivation

- Gain precision in a controlled way
  - Limited form of path-sensitivity, flow-sensitivity
    -
```
1 {s
2    if (multithreaded) {t fork(); t}
3   {t  ...  t}
4    if (multithreaded) {t lock(); t}
5   {t  ...  t}
6    if (multithreaded) {t unlock(); t}
7 s}
```

$$\{_t \ \ldots \{_s \textbf{ if true then } \{_t\ 5\ t\} \textbf{ else } \{_t\ \text{"foo"} + 3\ t\} s\} \ldots\ t\}$$

$$\{_t \ \ldots \{_s \textbf{ var } x = 1; \{_t\ \ldots\ t\}; x = \text{"foo"};\ s\} \ldots t\}$$

$$\{_t \ \ldots \{_s\ x{\rightarrow}obj = NULL;\ x{\rightarrow}obj = (\ldots)malloc(\ldots);\ s\} \ldots t\}$$

  - Even some context sensitivity

$$\{_s \textbf{ let } id\ x = x \textbf{ in} \{_t\ \ldots \{_s\ id\ 3\ s\}\ \ldots \{_s\ id\ 3.0\ s\}\ \ldots\ t\}\ s\}$$

Motivation

- Local refinement

```
{t
   let x : unknown int = ... in
   {s
      if x > 0 then {t (* x : pos int *) ...t}
      else if x = 0 then {t (* x : zero int *) ...t}
      else {t (* x : neg int *) ...t}
   s}
t}
```

```
{t
   {s
      x = (struct foo *) malloc(sizeof(struct foo));
      x→bar = ...;
      x→baz = ...;
      x→qux = ...;
   s}
   insert(shared_data_structure, x);
t}
```

Motivation

- Helping symbolic execution

```
{s
   let x ={t unknown_function() t}in ...
   let y ={t 2**z (* operation not supported by solver *)
  {t while true do{s loop_body s}t}
 s}
```

Formalism

Source Language.

e	$::= x \mid v$	variables, constants
	$\mid \; e + e$	arithmetic
	$\mid \; e = e \mid \neg e \mid e \wedge e$	predicates
	$\mid \; \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$	conditional
	$\mid \; \mathsf{let}\ x = e\ \mathsf{in}\ e$	let-binding
	$\mid \; \mathsf{ref}\ e \mid\ !e \mid e := e$	references
	$\mid \; \{_{\mathrm{t}}\ e\ _{\mathrm{t}}\}$	type checking block
	$\mid \; \{_{\mathrm{s}}\ e\ _{\mathrm{s}}\}$	symbolic execution block
v	$::= n \mid \mathsf{true} \mid \mathsf{false}$	concrete values

Types, Symbolic Expressions, and Environments.

τ	$::= \mathsf{int} \mid \mathsf{bool} \mid \tau\ \mathsf{ref}$	types
Γ	$::= \emptyset \mid \Gamma, x : \tau$	typing environment
s	$::= u{:}\tau$	typed symbolic expressions
g	$::= u{:}\mathsf{bool}$	guards
u	$::= \alpha \mid v$	symbolic variables, constants
	$\mid \; u{:}\mathsf{int} + u{:}\mathsf{int}$	arithmetic
	$\mid \; s = s \mid \neg g \mid g \wedge g$	predicates
	$\mid \; m[u{:}\tau\ \mathsf{ref}]$	memory select
m	$::= \mu$	arbitrary memory
	$\mid \; m, (s \rightarrow s)$	memory update
	$\mid \; m, (s \xrightarrow{\mathrm{a}} s)$	memory allocation
Σ	$::= \emptyset \mid \Sigma, x : s$	symbolic environment

Type checking

- Almost completely standard

$$\Gamma \vdash e : \tau$$

- Except for rule for $\{_s$ e $_s\}$… We'll come back to that

Symbolic execution

Symbolic Execution. $\boxed{\Sigma \vdash \langle S\,;\, e\rangle \Downarrow \langle S'\,;\, s\rangle \qquad S = \langle g\,;\, m\rangle}$

SEVAR

$$\frac{}{\Sigma, x : s \vdash \langle S\,;\, x\rangle \Downarrow \langle S\,;\, s\rangle}$$

SEVAL

$$\frac{}{\Sigma \vdash \langle S\,;\, v\rangle \Downarrow \langle S\,;\, (v{:}\,\mathrm{typeof}(v))\rangle}$$

SEPLUS

$$\frac{\Sigma \vdash \langle S\,;\, e_1\rangle \Downarrow \langle S_1\,;\, u_1{:}\mathsf{int}\rangle \qquad \Sigma \vdash \langle S_1\,;\, e_2\rangle \Downarrow \langle S_2\,;\, u_2{:}\mathsf{int}\rangle}{\Sigma \vdash \langle S\,;\, e_1 + e_2\rangle \Downarrow \langle S_2\,;\, (u_1{:}\mathsf{int} + u_2{:}\mathsf{int}){:}\mathsf{int}\rangle}$$

SEEQ

$$\frac{\Sigma \vdash \langle S\,;\, e_1\rangle \Downarrow \langle S_1\,;\, u_1{:}\tau\rangle \qquad \Sigma \vdash \langle S_1\,;\, e_2\rangle \Downarrow \langle S_2\,;\, u_2{:}\tau\rangle}{\Sigma \vdash \langle S\,;\, e_1 = e_2\rangle \Downarrow \langle S_2\,;\, (u_1{:}\tau = u_2{:}\tau){:}\mathsf{bool}\rangle}$$

SENOT

$$\frac{\Sigma \vdash \langle S\,;\, e_1\rangle \Downarrow \langle S_1\,;\, g_1\rangle}{\Sigma \vdash \langle S\,;\, \neg e_1\rangle \Downarrow \langle S_1\,;\, \neg g_1{:}\mathsf{bool}\rangle}$$

SEAND

$$\frac{\Sigma \vdash \langle S\,;\, e_1\rangle \Downarrow \langle S_1\,;\, g_1\rangle \qquad \Sigma \vdash \langle S_1\,;\, e_2\rangle \Downarrow \langle S_2\,;\, g_2\rangle}{\Sigma \vdash \langle S\,;\, e_1 \wedge e_2\rangle \Downarrow \langle S_2\,;\, (g_1 \wedge g_2){:}\mathsf{bool}\rangle}$$

SELET

$$\frac{\Sigma \vdash \langle S\,;\, e_1\rangle \Downarrow \langle S_1\,;\, s_1\rangle \qquad \Sigma, x : s_1 \vdash \langle S_1\,;\, e_2\rangle \Downarrow \langle S_2\,;\, s_2\rangle}{\Sigma \vdash \langle S\,;\, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\rangle \Downarrow \langle S_2\,;\, s_2\rangle}$$

SEIF-TRUE

$$\frac{\Sigma \vdash \langle S\,;\, e_1\rangle \Downarrow \langle S_1\,;\, g_1\rangle \qquad \Sigma \vdash \langle S_1[g \mapsto g(S_1) \wedge g_1]\,;\, e_2\rangle \Downarrow \langle S_2\,;\, s_2\rangle}{\Sigma \vdash \langle S\,;\, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \Downarrow \langle S_2\,;\, s_2\rangle}$$

SEIF-FALSE

$$\frac{\Sigma \vdash \langle S\,;\, e_1\rangle \Downarrow \langle S_1\,;\, g_1\rangle \qquad \Sigma \vdash \langle S[g \mapsto g(S_1) \wedge \neg g_1]\,;\, e_3\rangle \Downarrow \langle S_3\,;\, s_3\rangle}{\Sigma \vdash \langle S\,;\, \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rangle \Downarrow \langle S_3\,;\, s_3\rangle}$$

Symbolic execution of references

Symbolic Execution for References. $\boxed{\Sigma \vdash \langle S\,;\, e\rangle \Downarrow \langle S'\,;\, s\rangle}$

SEREF
$$\frac{\Sigma \vdash \langle S\,;\, e_1\rangle \Downarrow \langle S_1\,;\, u_1{:}\tau\rangle \qquad \alpha \notin \Sigma, S, S_1, u_1 \qquad S' = S_1[m \mapsto (m(S_1), (\alpha{:}\tau\ \mathsf{ref} \xrightarrow{\mathrm{a}} u_1{:}\tau))]}{\Sigma \vdash \langle S_1\,;\, \mathsf{ref}\ e_1\rangle \Downarrow \langle S'\,;\, \alpha{:}\tau\ \mathsf{ref}\rangle}$$

SEASSIGN
$$\frac{\Sigma \vdash \langle S\,;\, e_1\rangle \Downarrow \langle S_1\,;\, s_1\rangle \qquad \Sigma \vdash \langle S_1\,;\, e_2\rangle \Downarrow \langle S_2\,;\, s_2\rangle}{\Sigma \vdash \langle S\,;\, e_1 := e_2\rangle \Downarrow \langle S_2[m \mapsto (m(S_2), (s_1 \to s_2))]\,;\, s_2\rangle}$$

SEDEREF
$$\frac{\Sigma \vdash \langle S\,;\, e_1\rangle \Downarrow \langle S_1\,;\, u_1{:}\tau\ \mathsf{ref}\rangle \qquad \vdash m(S_1)\ \mathsf{ok}}{\Sigma \vdash \langle S\,;\, !e_1\rangle \Downarrow \langle S_1\,;\, m(S_1)[u_1{:}\tau\ \mathsf{ref}]{:}\tau\rangle}$$

How to ensure that dereference is of appropriate type?

Note: in assignment of s_2 to $\alpha{:}\tau$ ref does not require s_2 to be of type τ

Memory Type Consistency. $\boxed{\vdash m\ \mathsf{ok}\ U \qquad \vdash m\ \mathsf{ok}}$

EMPTY-OK
$$\frac{}{\vdash \mu\ \mathsf{ok}\ \emptyset}$$

ALLOC-OK
$$\frac{\vdash m\ \mathsf{ok}\ U}{\vdash m, (\alpha{:}\tau\ \mathsf{ref} \xrightarrow{\mathrm{a}} u_2{:}\tau)\ \mathsf{ok}\ U}$$

OVERWRITE-OK
$$\frac{\vdash m\ \mathsf{ok}\ U \quad U' = U \setminus \{s_1 \to s_2 \mid s_1 \equiv u_1{:}\tau\ \mathsf{ref} \wedge s_1 \to s_2 \in U\}}{\vdash m, (u_1{:}\tau\ \mathsf{ref} \to u_2{:}\tau)\ \mathsf{ok}\ U'}$$

ARBITRARY-NOTOK
$$\frac{\vdash m\ \mathsf{ok}\ U}{\vdash m, (s_1 \to s_2)\ \mathsf{ok}\ (U \cup \{s_1 \to s_2\})}$$

M-OK
$$\frac{\vdash m\ \mathsf{ok}\ \emptyset}{\vdash m\ \mathsf{ok}}$$

Design decisions

- Deferral vs. execution
  - When to execute (e.g., forking on an if) versus deferring (e.g., having a _?_:_ symbolic operator)
- Tension between symbolic execution and types
  - Want symbolic execution to be permissive, but need to have enough information around to invoke type checking
  - $\vdash m(S_1)$ ok requires entire memory to be appropriately typed at time of dereference
  - Alternatives:
    - Fork execution for each possible actual address pointer could evaluate to
    - Use external alias analysis to ensure points-to set is well-typed

Mixing

Block Typing.

$\boxed{\Gamma \vdash e : \tau}$

TSymBlock

$$\frac{\begin{array}{c}\Sigma(x) = \alpha_x{:}\Gamma(x) \qquad (\text{for all } x \in \mathit{dom}(\Gamma)) \\ \Sigma \vdash \langle S\,;\, e\rangle \Downarrow \langle S_i\,;\, u_i{:}\tau\rangle \qquad S = \langle \mathsf{true}\,;\, \mu\rangle \qquad \mu \notin \Sigma \\ \vdash m(S_i)\ \mathsf{ok} \qquad \mathit{exhaustive}(g(S_1), \ldots, g(S_n)) \qquad (i \in 1..n)\end{array}}{\Gamma \vdash \{_{\mathrm{s}}\ e\ _{\mathrm{s}}\} : \tau}$$

Symbolic execution input

Symbolic execution output

$$\mathit{exhaustive}(g_1, \ldots, g_n) \iff (g_1 \vee \ldots \vee g_n \text{ is a tautology})$$

Block Symbolic Execution.

$\boxed{\Sigma \vdash \langle S\,;\, e\rangle \Downarrow \langle S'\,;\, s\rangle}$

SETypBlock

$$\frac{\vdash \Sigma : \Gamma \qquad \vdash m(S)\ \mathsf{ok} \qquad \Gamma \vdash e : \tau \qquad \mu', \alpha \notin \Sigma, S}{\Sigma \vdash \langle S\,;\, \{_{\mathrm{t}}\ e\ _{\mathrm{t}}\}\rangle \Downarrow \langle S[m \mapsto \mu']\,;\, \alpha{:}\tau\rangle}$$

Symbolic and Typing Environment Conformance.

$\boxed{\vdash \Sigma : \Gamma}$

$$\frac{\begin{array}{c}\mathit{dom}(\Sigma) = \mathit{dom}(\Gamma) \\ \Sigma(x) = u{:}\Gamma(x) \qquad (\text{for all } x \in \mathit{dom}(\Gamma))\end{array}}{\vdash \Sigma : \Gamma}$$

Soundness

- Yes, it's sound.
- Proof by simultaneous induction for type soundness and "symbolic execution soundness"
  - Need to define a soundness relation between concrete state and symbolic state

Mixy

- Tool for C for detecting null pointer errors
- Uses a type qualifier system
  - `τ *nonnull` means pointer can never be null
  - `τ *null` means pointer can never be null
- Performs **inference** instead of checking
  - Generates symbolic variables for unknown qualifiers, and generates equality constraints over these variables

```
1 void free(int *nonnull x);
2 int *id(int *p) { return p; }
3 int *x = NULL;
4 int *y = id(x);
5 free(y);
```

$$\text{free} : \text{int} * \text{nonnull} \rightarrow \text{void} \qquad x : \text{int} * \beta$$

$$\text{id} : \text{int} * \gamma \rightarrow \text{int} * \delta \qquad y : \text{int} * \epsilon$$

$$\text{null} = \beta \quad \beta = \gamma \quad \gamma = \delta \quad \delta = \epsilon \quad \epsilon = \text{nonnull}$$

- Methodology: start with all type-checking, and identify false positives. Add symbolic blocks lazily to improve precision

Mixy

- Functions are declared to be either typed or symbolic
  - Can get block-level mixing by refactoring
- Translating between type-checking and symbolic execution
  - Type checking uses symbolic variables for qualifiers
  - If x has type `int *null`, symbolic environment initializes x to (α:bool)?*loc*:0
  - If x has type `int *`β, (i.e., unknown qualifier) then assume that β=`nonnull`
    - If wrong, will need to redo symbolic execution, until fixpoint reached

Aliasing

- First performs a pointer analysis to discover aliasing relationships
- When going from symbolic state to typing environment, check that all pointers within a points-to set have same type
  - Required for soundness, analogous to $\vdash M(S)$ ok
  - Major performance bottle neck
  - Also source of imprecision (since pointer analysis is context-insensitive)

Caching and recursion

- Cache results of symbolic execution and type checking
  - Use type context to summarize blocks
- Typed block and symbolic block may recursively call each other
  - Need to prevent infinite recursion
  - Maintain stack of what blocks are currently being analyzed, prevent infinite recursion
  - Will require iteration until fix-point reached