Saturn

Lecture 3
CS357

Goals

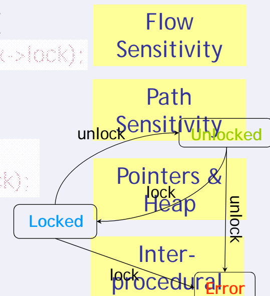
- Scale to the largest programs
- Be precise
  - *path-sensitive & context-sensitive*
  - *model most operations exactly*
- Back in 2004, this hadn't been done.

Limitations

- Give up on soundness
  - *At least temporarily*
- Why?

Code Example

```
void f(state *x, state *y) {
  result = spin_trylock(&x->lock);
  spin_lock(&y->lock);
  ...
  if (!result)
      spin_unlock(&x->lock);
  spin_unlock(&y->lock);
}
```

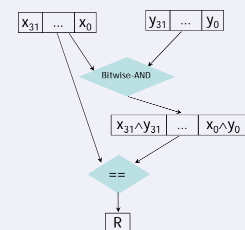


Saturn

- What?
  - *SAT-based approach to static bug detection*
- How?
  - *Program constructs ➔ Boolean constraints*
  - *Inference ➔ SAT solving*
- Why SAT?
  - *Program states naturally expressed as bits*
  - *The theory for bits is SAT*
  - *Efficient solvers available*

Straight-Line Code

```
void f(int x, int y)
{
  int z = x & y ;
  assert(z == x);
}
```



Straight-line Code

```
void f(int x, int y)
{
  int z = x & y;
  assert(z == x);
}
```

Query: Is-Satisfiable(¬)

Answer: Yes

x = [00...1] y = [00...0]

Negated assertion is satisfiable.

Therefore, the assertion may fail.

R

Primitive Operations

- What language primitives can be represented by boolean functions?
- All of them!
  - *Every finite function is equivalent to some boolean expression*

Easy Ones

- Logical operations
  - *&, |, ~, bit shifts*
- Relational operations
  - *==, >, <=, !*

Slightly Harder

- Addition
  - *Chain 32 1-bit adders together*
- Multiplication by a constant k
  - *Chain k 32-bit adders together*

1-bit adder

Add a to b with carry c

Bit: (a xor b) xor c

Carry: $(a \wedge b) \vee (b \wedge c) \vee (a \wedge c)$

Casts

- Casts don't change the value of any bits
- But
  - *May increase or decrease the number of bits*
    - *Widening casts pad*
    - *Drop appropriate bits for narrowing casts*
- Casts are easy in this representation
  - *But a major problem for most C analyzers.*

Too Hard

- Some operations are too complicated
  - *Multiplication of variables*
  - *Division, mod*
  - *Floating point operations*
- Model by introducing unconstrained boolean variables for the result
  - *Represents unknown outcome*

Summary

- Model primitive operations as boolean functions
- Most can be modeled exactly
  - *No loss of information*

Control Flow

```
if (c)
   x = a;          G = c, x: [a31...a0]
else
   x = b;          G = ¬c, x: [b31...b0]
res = x;           G = true , x: [v31...v0]
```

where $v_i = (c \wedge a_i) \vee (\neg c \wedge b_i)$

- Merges
  - *preserve path sensitivity*
  - *select bits based on the values of incoming guards*

Loops

- Loops not allowed
  - *Unroll loops k times, drop backedges*
- May miss errors that are deeply buried
  - *Bug finding, not verification*
  - *Hypothesis: Many errors surface in a few iterations*
- Advantages
  - *Simplicity, reduces false positives*

Pointers – Overview

- May point to different locations…
  - *Thus, use points-to sets*
    $p: \{ l_1, \ldots, l_n \}$
- … but path sensitive
  - *Use guards on points-to relationships*
    $p: \{ (g_1, l_1), \ldots, (g_n, l_n) \}$

Pointers – Example

```
➡ p = &x;           G = true, p: { (true, x) }
  if (c)
➡    p = &y;
➡ res = *p;        G  if (c) res = y;
                       else if (¬c) res = x;     x)}
```


Pointers – Recap

- Guarded Location Sets
  $\{ (g_1, l_1), \ldots, (g_n, l_n) \}$
- Guards
  - *Condition under which points-to relationship holds*
  - *Collected from statement guards*
- Pointer Dereference
  - *Conditional assignments*

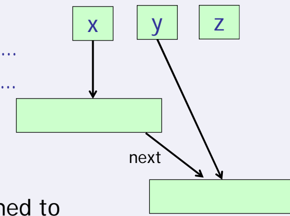
Modeling the Environment

```
f(foo *x) {
    ... *x ...
}
```

- What do we get when x is dereferenced?
  - *Build environment lazily.*
  - *When x is dereferenced, create new object (new bits) of the appropriate type*

Modeling the Environment

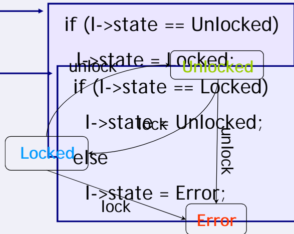
```
F(foo *x) {
    ... y = x->next ...
    ... z = y->next ...
}
```



- All environment objects are assumed to be unaliased.

What can we do with Saturn?

```
int f(lock_t *l)
{
    lock(l);
    ...
    unlock(l);
}
```



General FSM Checking

- Encode FSM in the program
  - *State ➔ Integer*
  - *Transition ➔ Conditional Assignments*
- Check code behavior
  - *SAT queries*

How are we doing so far?

- Precision: ☺
- Scalability: ☹
  - *SAT limit is 1M clauses*
  - *About 10 functions*
- Solution:
  - *Divide and conquer*
  - *Function summaries*

Function Summaries (1st try)

```
int f(lock_t *l)
{
    lock(l);
    ...

    ...
    unlock(l);
    return 0;
}
```

- Function behavior can be summarized with a set of state transitions
- Summary:
  *l: Unlocked → Unlocked
  Locked → Error

Computing Function Summaries

```
int f(lock_t *l)
{
   lock(l);
   …

   …
   unlock(l);
   return 0;
}
```

SAT(
  Formula(body of f)
  $\wedge$ (*l starts unlocked)
  $\wedge$ (*l ends unlocked)
)

Profs. Aiken & Dill CS357 Lecture 3 25

Computing Function Summaries

SAT(
  Formula(body of f)
  $\wedge$ (loc l starts in state x)
  $\wedge$ (loc l ends in state y)
)
for every combination of l, x, and y in f

State transition exists if satisfiable

Profs. Aiken & Dill CS357 Lecture 3 26

A Difficulty

```
int f(lock_t *l)
{
   lock(l);
   …
   if (err) return -1;
   …
   unlock(l);
   return 0;
}
```

- Problem
  - *two possible output states*
  - *distinguished by return value*
    (retval == 0)…
- Summary
  1. (retval == 0)
     *l: Unlocked → Unlocked
     Locked → Error
  2. ¬(retval == 0)
     *l: Unlocked → Locked
     Locked → Error

Profs. Aiken & Dill CS357 Lecture 3 27

FSM Function Summaries

- Summary representation (simplified):
  $\{ P_{in}, P_{out}, R \}$
- User gives:
  - P_{in}: *predicates on initial state*
  - P_{out}: *predicates on final state*
  - *Express interprocedural path sensitivity*
- Saturn computes:
  - *R: guarded state transitions*
  - *Used to simulate function behavior at call site*

Profs. Aiken & Dill CS357 Lecture 3 28

Lock Summary (2nd try)

```
int f(lock_t *l)
{
   lock(l);
   …
   if (err) return -1;
   …
   unlock(l);
   return 0;
}
```

- Output predicate:
  - $P_{out} = \{ (retval == 0) \}$
- Summary (R):
  1. (retval == 0)
     *l: Unlocked → Unlocked
     Locked → Error
  2. ¬(retval == 0)
     *l: Unlocked → Locked
     Locked → Error

Profs. Aiken & Dill CS357 Lecture 3 29

Scalability

- How big can a summary based on FSM be?
  - *# of FSM state pairs*
  - *times 2^(# of predicates)*
  - *times # of modeled locations*
  - *Note: Not dependent on size of the code!*
- Locking example:
  - *3 states, 1 predicate, 1 location =*
    *(3 * 2) * 2 * 1 = 12 SAT tests*

Profs. Aiken & Dill CS357 Lecture 3 30

Lock Checker for Linux

- Parameters:
  - *States: { Locked, Unlocked, Error }*
  - *P_{in} = {}*
  - *P_{out} = { (retval == 0) }*
- Experiment:
  - *Linux Kernel 2.6.5: 4.8MLOC*
  - *~40 lock/unlock/trylock primitives*
  - *20 hours to analyze*
    - *3.0GHz Pentium IV, 1GB memory*

Double Locking/Unlocking

```
static void sscape_coproc_close(…) {
➡ spin_lock_irqsave(&devc->lock, flags);
  if (…)
➡     sscape_write(devc, DMAA_REG, 0x20);
  …
}

static void sscape_write(struct … *devc, …) {
➡ spin_lock_irqsave(&devc->lock, flags);
  …
}
```


Ambiguous Return State

```
int i2o_claim_device(…) {
➡ down(&i2o_configuration_lock);
  if (d->owner) {
   ➡ up(&i2o_configuration_lock);
   ➡ return -EBUSY;
  }
  if (…) {
   ➡ return -EBUSY;
  }
  …
}
```


Bugs

Type	Bugs	False Pos.	% Bugs
Double Locking	134	99	57%
Ambiguous State	45	22	67%
Total	179	121	60%

Previous Work: MC (31), CQual (18), <20% Bugs

Function Summary Database

- 63,000 functions in Linux
  - *More than 23,000 are lock related*
  - *17,000 with locking constraints on entry*
  - *Around 9,000 affect more than one lock*
  - *193 lock wrappers*
  - *375 unlock wrappers*
  - *36 with return value/lock state correlation*

The Rest

- Talk about some details
  - *Strengths and weaknesses*
- A few general principles
  - *Illustrated using Saturn*

Bailing Out

- It is always useful to understand how an analysis gives up
- In Saturn, unanalyzable expressions are modeled by fresh bit variables
  - *Makes constraints easier to satisfy*

Loop Unrolling

- Unrolling a few iterations of loops pitched as a feature
  - *Bounded model-checking work does the same*
- But this is not just convenient
  - *It's necessary!*
  - *No good way to express feedback constraints of loops/recursive functions when variables range only over 0 and 1*

Function Side-Effects

- To be sound(er), must take into account side-effects of functions

```
f(x) {  … g(y) … }
    g(z) { … *z = w … }
```

- Saturn tracks the set of locations modified by a function
  - *Modified locations set to unknown bits in the caller*

Conditionals

- There is a problem with conditionals

  if (p) then s_1 else s_2

- Assume the guard for s_1 is g_p
- What is the guard for s_2?
  - *if g_p is exact, then it is $\neg g_p$*
  - *but if g_p is an overapproximation, then it is not $\neg g_p$*

The Heap

- Saturn is very strong at reasoning about
  - *Control flow*
  - *Local variables*
  - *Expressions*
- But it is weak at deep heap reasoning
  - *Assumes no aliasing in environment*
  - *Can't really express recursive data structures*

Optimization

- Saturn builds a formula for every bit used in a function body
  - *But not all are used in queries!*
- Example

```
x = y * z
a = b + c
assert(x == 0)
```

- This optimization is important!
  - *Big speedup in solving*

Lesson 1

Contrary to popular belief, there is no magic in solvers.

Smaller formulas take less time to solve.

Lesson 2

Be as lazy as possible!

Work delayed long enough is never done.

Lesson 3

Scalability requires bounding the size of formulas.

Space is the critical resource, not time.

Lesson 4

- Analyzing in one direction is problematic
  - *Forwards or backwards*
- Constraints
  - *Give a global picture of the program*
  - *Allow more efficient order of solution*

Why SAT? (Revisited …)

- Moore's Law
- Uniform modeling of constructs as bits
- Constraints
  - *Local specification*
  - *Global solution*
- Incremental SAT solving
  - *makes multiple queries efficient*

Why SAT? (Cont.)

- Path sensitivity is important
  - *To find bugs*
  - *To reduce false positives*
  - *Much easier to model precisely with SAT*
- Compositionality is important
  - *Function summaries critical for scalability*
  - *Easy to construct with SAT queries*