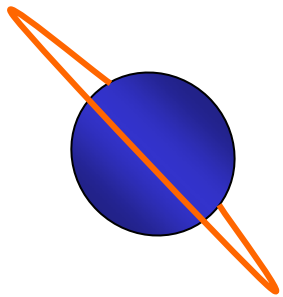
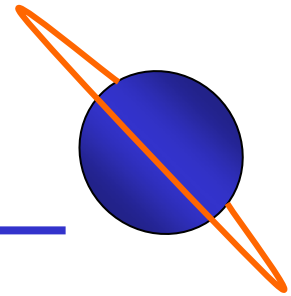
Scalable Program Analysis Using Boolean Satisfiability:

The Saturn Project

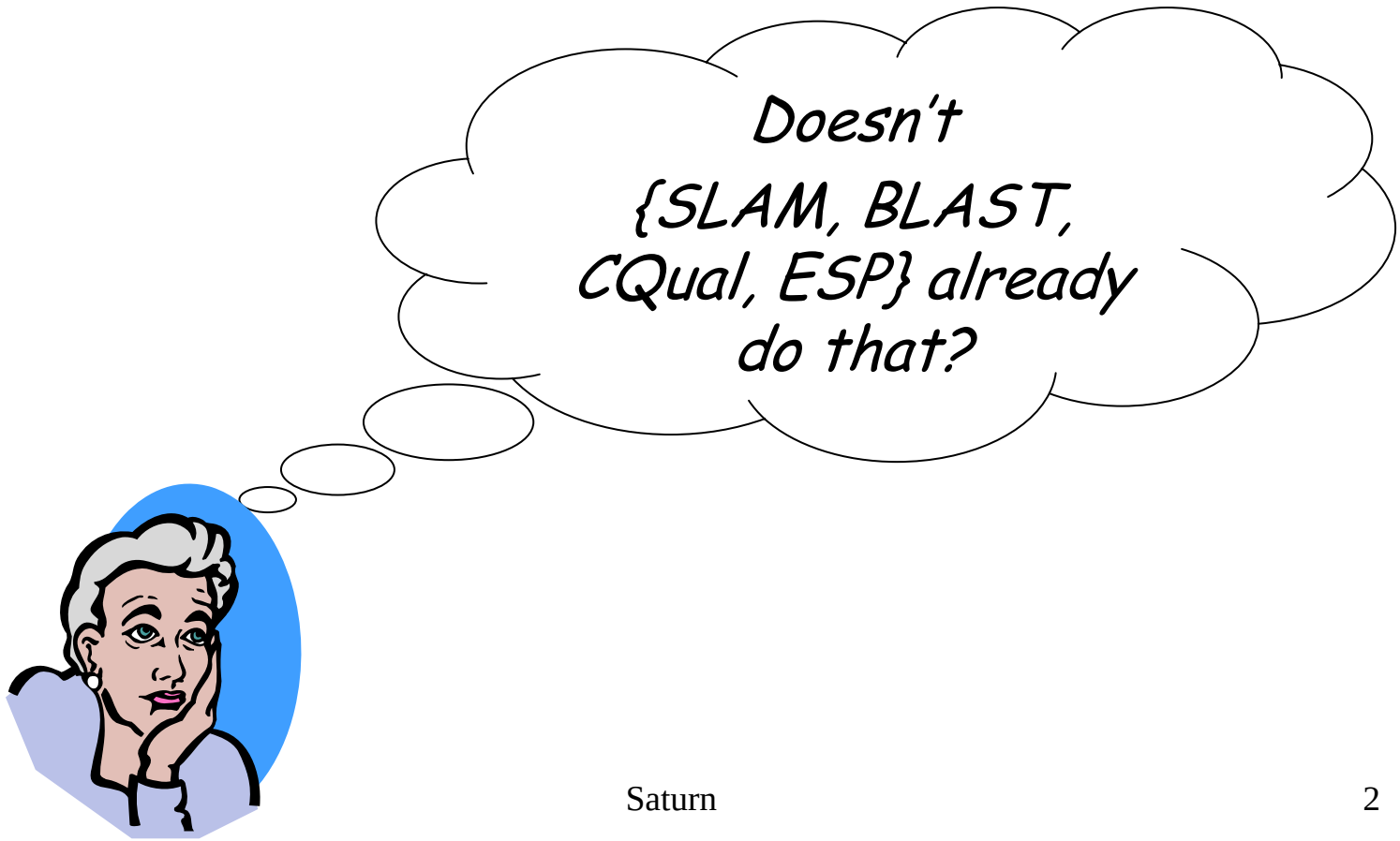


Alex Aiken
Stanford University

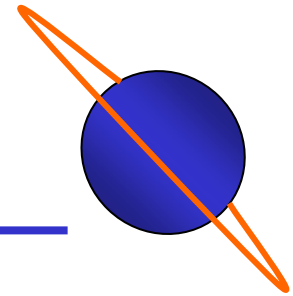
The Idea



Verify properties of large systems!

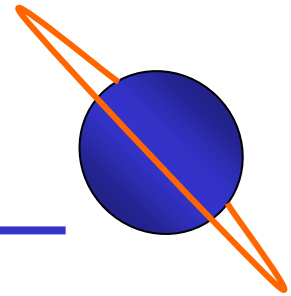


Well, No . . .



- Some systems work on large programs
  - Millions of lines of code
- Some systems verify properties
  - E.g., alias-aware type state
- But none do both
  - That I know of . . .

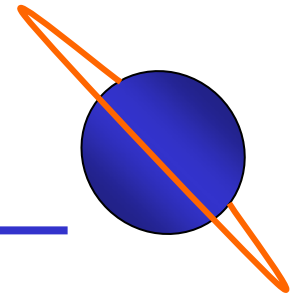
Scaling vs. Precision



- Scaling
  - Need to handle multi-million line programs
  - Why?
    - Because that is where automatic analysis does the most good
    - Because they are there
  - Pushes towards low-complexity algorithms

- Precision
  - High degree of automation a requirement
    - Little user input (few annotations)
    - Efficient to use output (few spurious warnings)
  - Pushes towards high-complexity algorithms

Set-up For A Story . . .



- Alias Analysis
  - Basic to verification
  - Paradigmatic problem

```
*x = ...
...
... = *y
```

*Can *x and *y be aliases?*

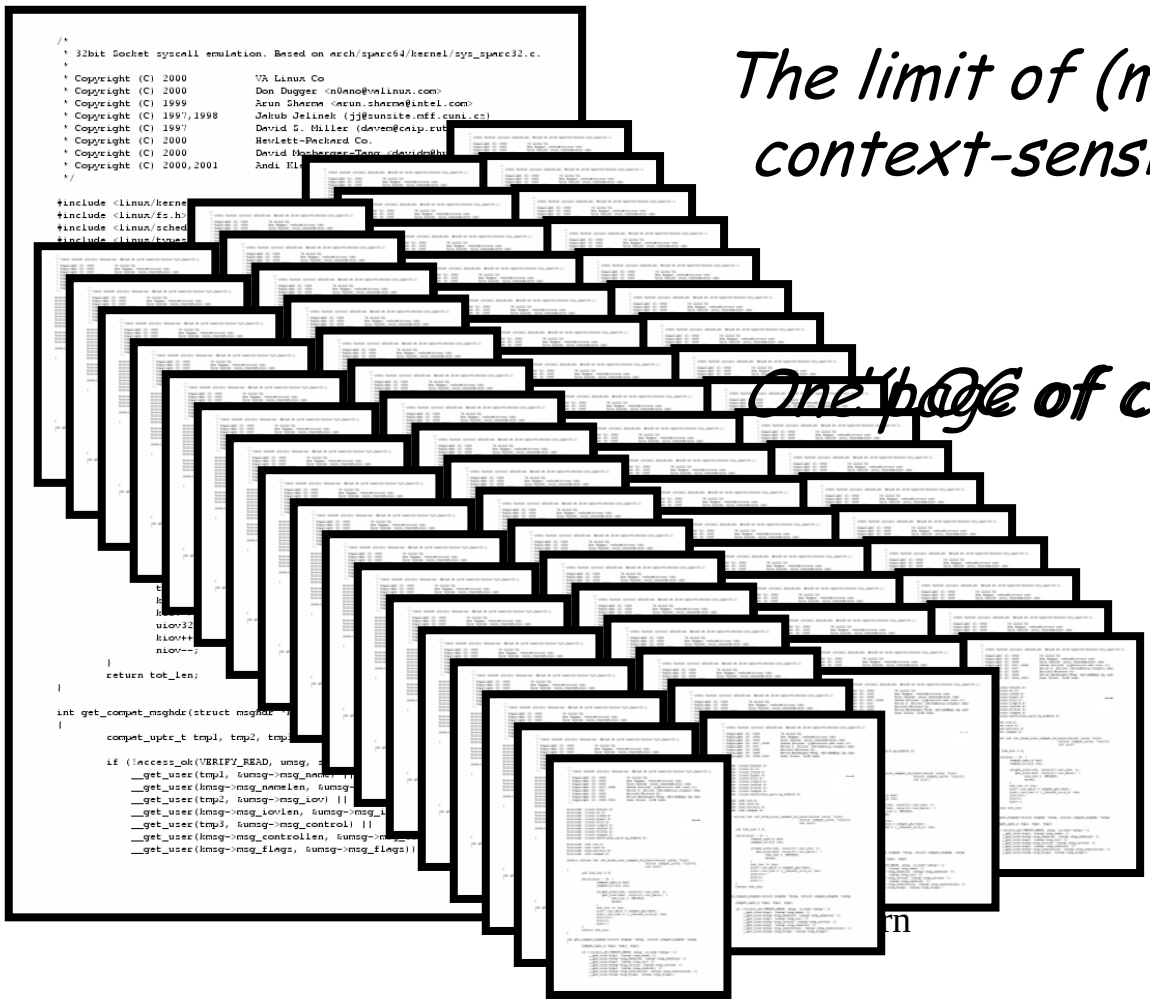
- Dimensions of precision
  - * = *{sensitive,insensitive}*
- *Flow-**

```
X = 1;
Y = X + 1;
```

- *Context-**

```
F() { ... H() ...}
G() { ... H() ...}
```

A Parable About Alias Analysis

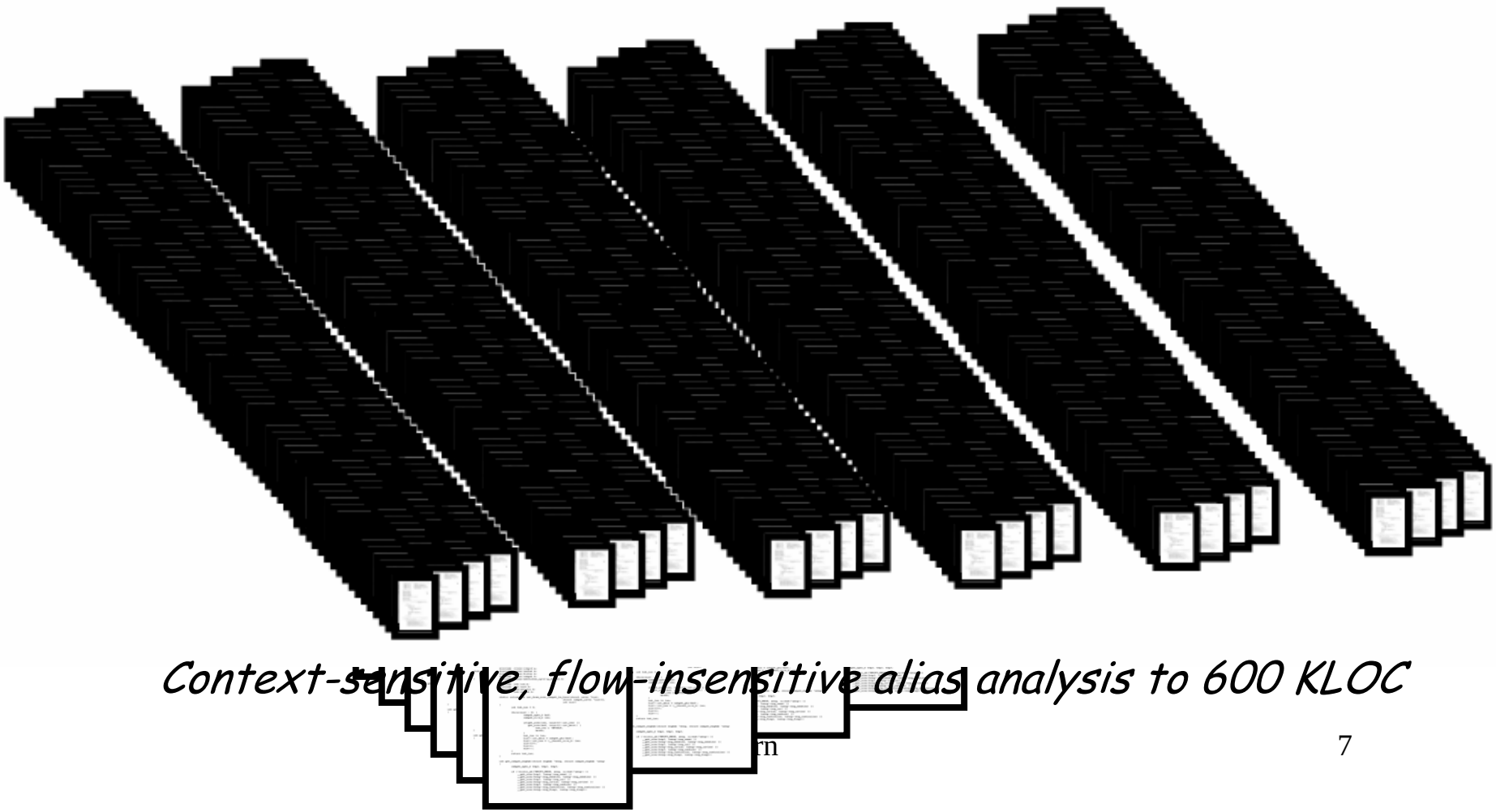


Four KLOC of code from Linux . . .

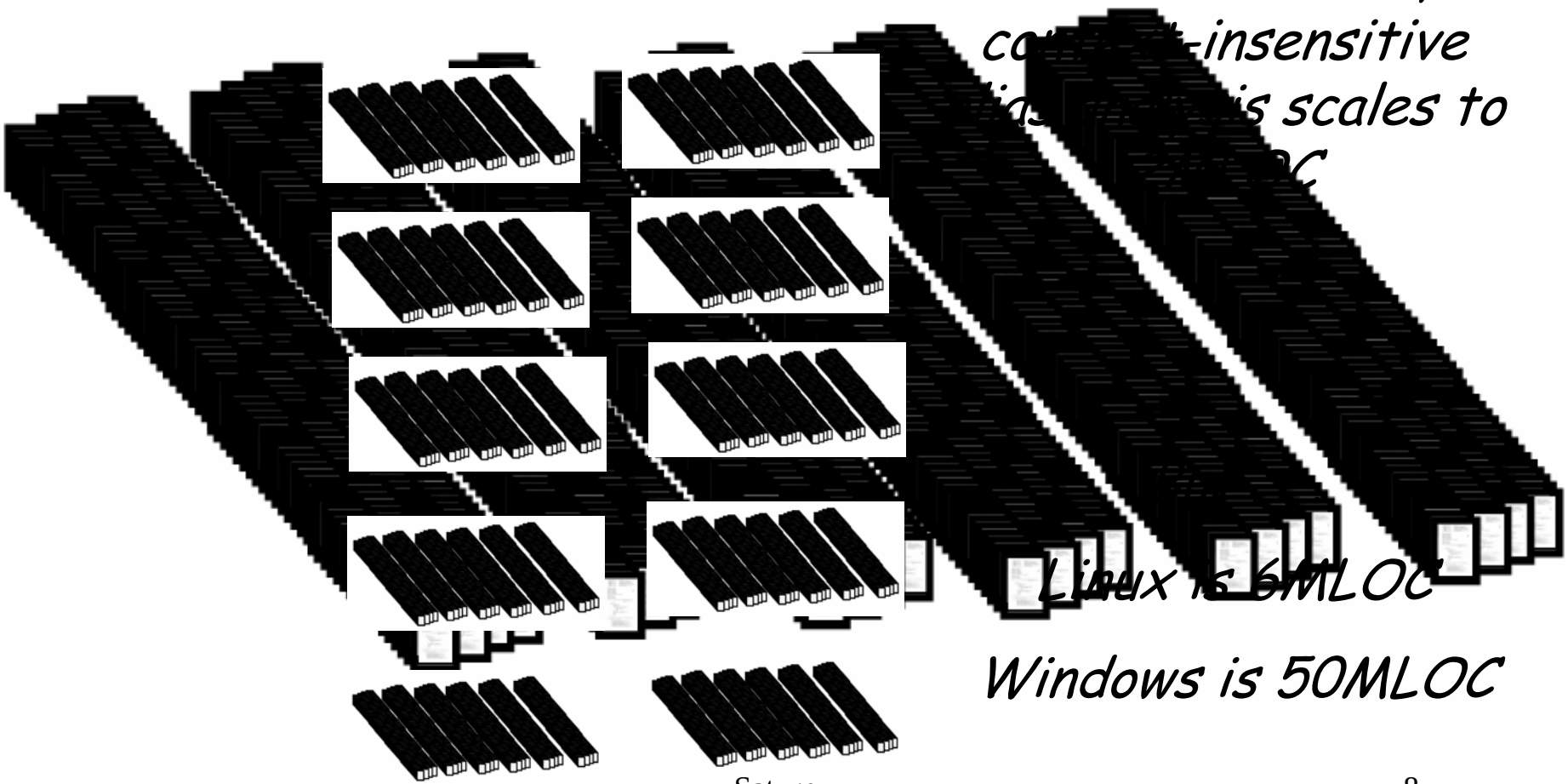
The limit of (most) flow-sensitive, context-sensitive alias analyses.

One page of code from Linux . . .

A Parable Continued



A Parable Continued

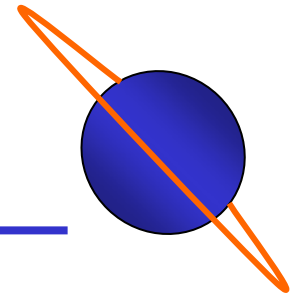


Flow-insensitive, co[illegible]-insensitive [illegible]is scales to [illegible]C

Linux is 6MLOC

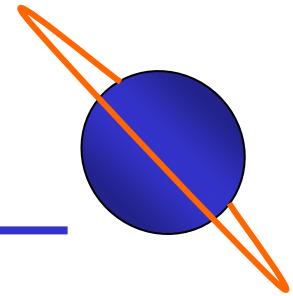
Windows is 50MLOC

This Talk



- An approach to achieving both precision and scalability
  - Based on SAT and other constraint solvers
- Some examples
  - A sound alias analysis
  - Unsound null dereference analysis
  - Type state (if time)

The Main Idea

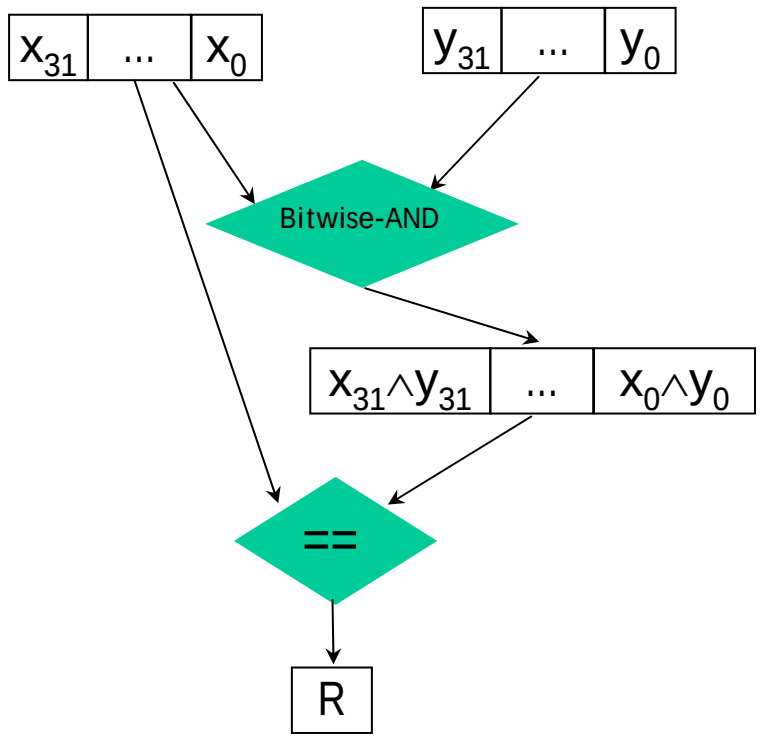


- For precision, delay abstraction
  - Model function/loop bodies very precisely
  - (Almost) no abstraction intraprocedurally

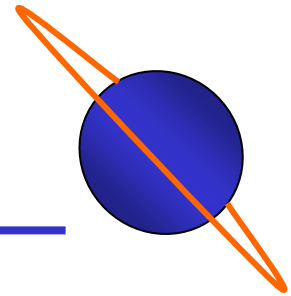
- For scalability, abstract at function boundaries
  - Summarize a function's behavior
  - Summaries designed per property
    - Analysis design = summary design
  - Intuition: Programmers also abstract at these boundaries

Procedure Bodies: Bit Blast

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



Highlights



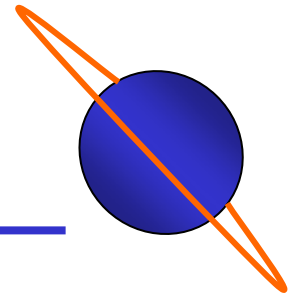
- Compute a guard for each statement
  - Based on e.g., predicates in conditionals
- Loops are tail-recursive functions
- Pointers
  - May point to different locations…
  - Thus, use points-to sets

    $p: \{ l_1, \ldots, l_n \}$

  - … but path sensitive
  - Use guards on points-to relationships

    $p: \{ (g_1, l_1), \ldots, (g_n, l_n) \}$

Pointers – Example



- p = &x;

  if (c)

- p = &y;

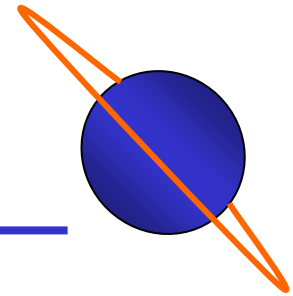
- res = *p;

G = true, p: { (true, x) }

```
if (c) res = y;
else if (¬c) res = x;
```

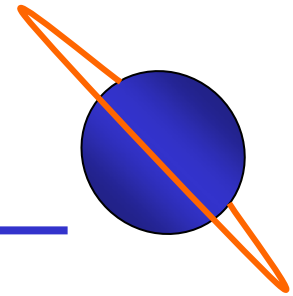
G , x)}

Other Stuff



- Other constructs
  - Structs, …
- Modeling of the environment
- Optimizations
  - several to reduce size of formulas

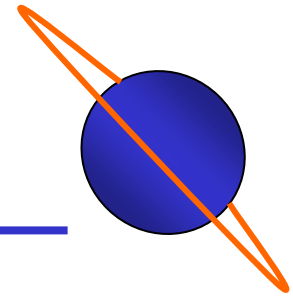
Summary



- Compile code into boolean circuits
  - Very accurate representation
  - Works great if your program is < 500 lines of code

- Related work
  - Bit-blasting well-known in model-checking
    - Clarke & Kroening
  - Work in software architecture
    - Alloy project at MIT

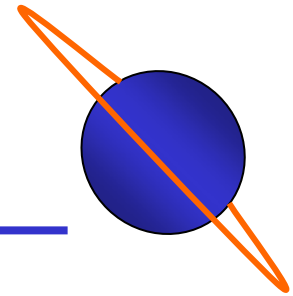
Two Questions



- What can we use this approach for?

- How can it scale?

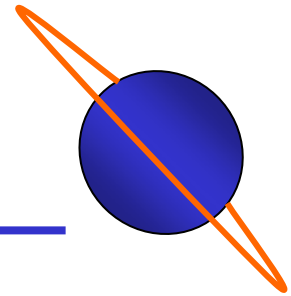
Example: Alias Analysis



- Illustrate with a sound, scalable alias analysis
  - For C

- Needed for almost any interesting verification problem

Points-to Rule

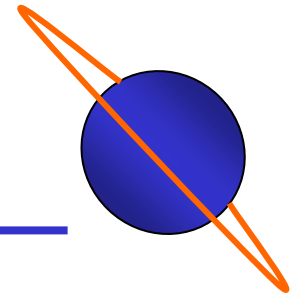


- PointsTo(p, l)
  - Condition under which p points to l
  - *A guarded points-to graph*

$$\psi(p) = \{ (g_0, l_0), \ldots, (g_{n-1}, l_{n-1}) \}$$

$$\text{PointsTo}(p, l) = \begin{cases} g_i & (\text{if } l_i = l) \\ \\ \text{false} & (\text{otherwise}) \end{cases}$$

Function Summaries

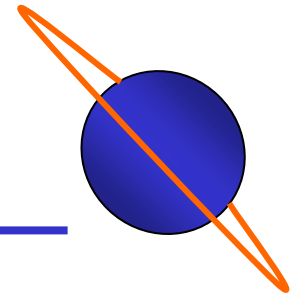


- For a function f
  - Given an entry points-to graph P_{in}
  - Compute an exit points-to graph P_{out}

- f's summary is then the signature

$$P_{in} \,!\, P_{out}$$

Context-Sensitivity



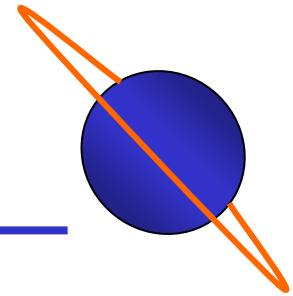
- Signature for f in terms of names visible in f
  - Parameter and global variable names

- Consider function f(a,b) w/summary P_{in} ! P_{out}

- At call site f(a',b')
  - Compute substitution of actual for formal names

    [a -> a', b -> b']
  - Call adds points-to relations P_{out} [a -> a', b -> b']

Termination and Soundness

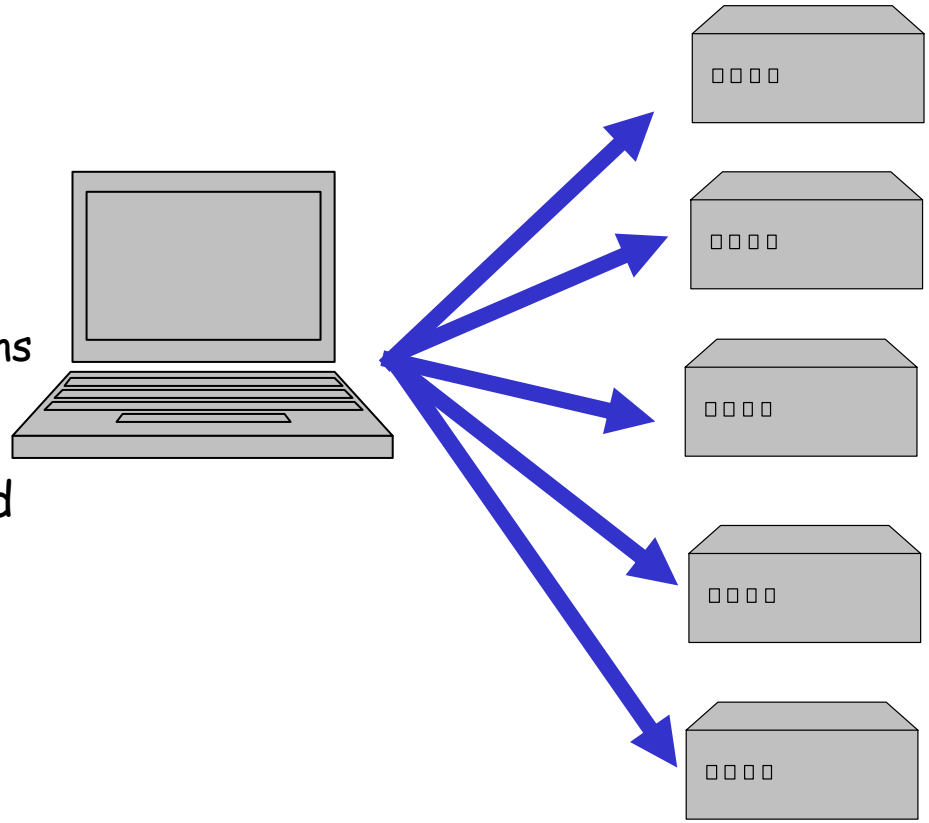


- All guards in summaries are true/false
  - At function exit, promote satisfiable guards to true
  - Clearly sound

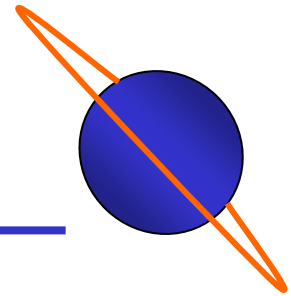
- Begin with empty summaries for all functions
  - # of graph nodes < # of accesses in function
  - Edges are only added, never removed
  - Together, implies termination

Alias Analysis Results

- Parallel implementation
  - Server sends functions to clients to analyze
  - Used by all analyses, not just alias analysis
- Analyze all of Linux in 1 hr 20 min on 40 cores
  - 6MLOC
  - Analysis of about 1% of functions times out
- Interprocedurally context- and object-senstive
- Intraprocedurally flow- and path-sensitive

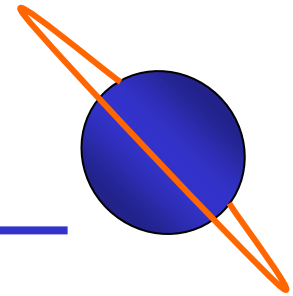


Study of Aliasing in 1MLOC



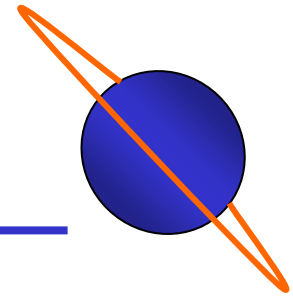
- Almost all aliasing falls into one of 8 categories
  - Parent pointers
  - Child pointers
  - Shared read pointers
  - One reader/one writer
  - 4 kinds of index cursors
- 20% false aliasing
- Outside of heap data structures & globals, aliasing is rare
  - 2.4% of functions use other aliased values
- Found unintentional aliasing causing subtle bug in PostgreSQL

Why Does It Work?



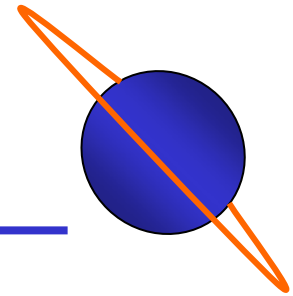
- Good match to programmer thinking
- Complex invariants within a function
  - No or little abstraction
- Simpler interface between functions
  - Per-property abstraction
- Summarization at function boundaries exploits abstraction

Why Does It Work?



- Good match to computer systems
- Analyze one function at a time
  - Only one function in RAM
  - Summaries for others in disk database
- Easily parallelized

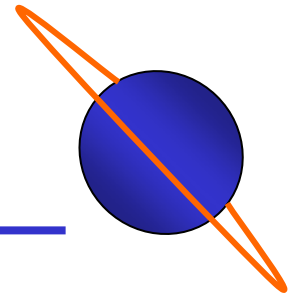
An Application: NULL analysis



- NULL pointer dereferences cause crashes
  - In C
  - Exceptions in safe languages

- Common, if low-level, programming error

Inconsistencies

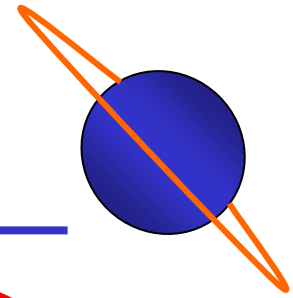


- Look for *inconsistency errors*

- Pointer is dereferenced in two places
  - In one place it is checked for NULL
  - In the other place it is not

- Empirically, very likely a bug
  - Instead of a redundant check
  - Note this test cannot catch all NULL errors

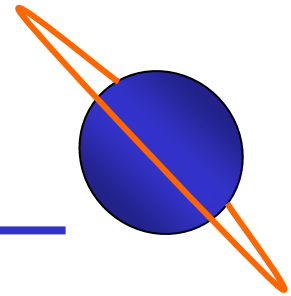
Example



```
680     struct usb_tt   *tt = urb->dev->tt;
...
696     think_time = tt ? tt->think_time : 0;
...
736      if (!ehci_is_TDI(ehci)
                 || urb->dev->tt->hub !=  . . .
```

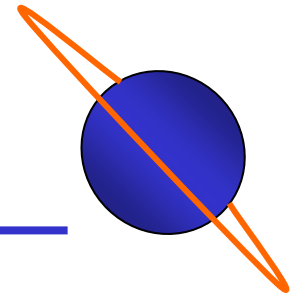
Must deal with aliasing . . .

Formalization of the Problem



- The problem has two parts
- When are two pointers "the same"?
- Given two pointers that are the same, is one checked for NULL and the other not?

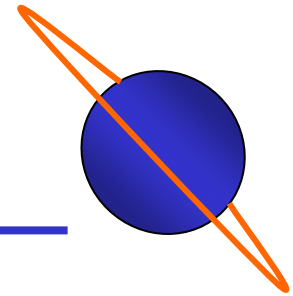
Part I



- Pointers x and y are the same if

$$8\, l.\ \mathrm{PointsTo}(x,l)\ ,\ \mathrm{PointsTo}(y,l)$$

Part II

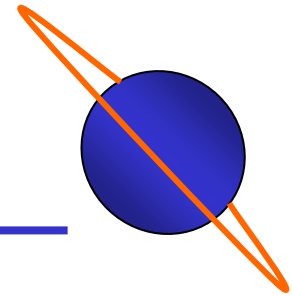


- Consider
  - Pointer x at statement s_1 with statement guard g_1
  - Pointer y at statement s_2 with statement guard g_2

- If x and y are the same and

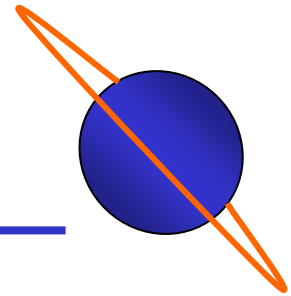
$$(g_1 \,!: \text{PointsTo}(x,\text{NULL}))$$
$$Æ$$
$$:(g_2 \,!: \text{PointsTo}(y,\text{NULL}))$$

Comments



- The definition is purely semantic
  - No special cases
  - No pattern matching on (x == NULL)
  - etc.
- Also concise
- And finds bugs . . .

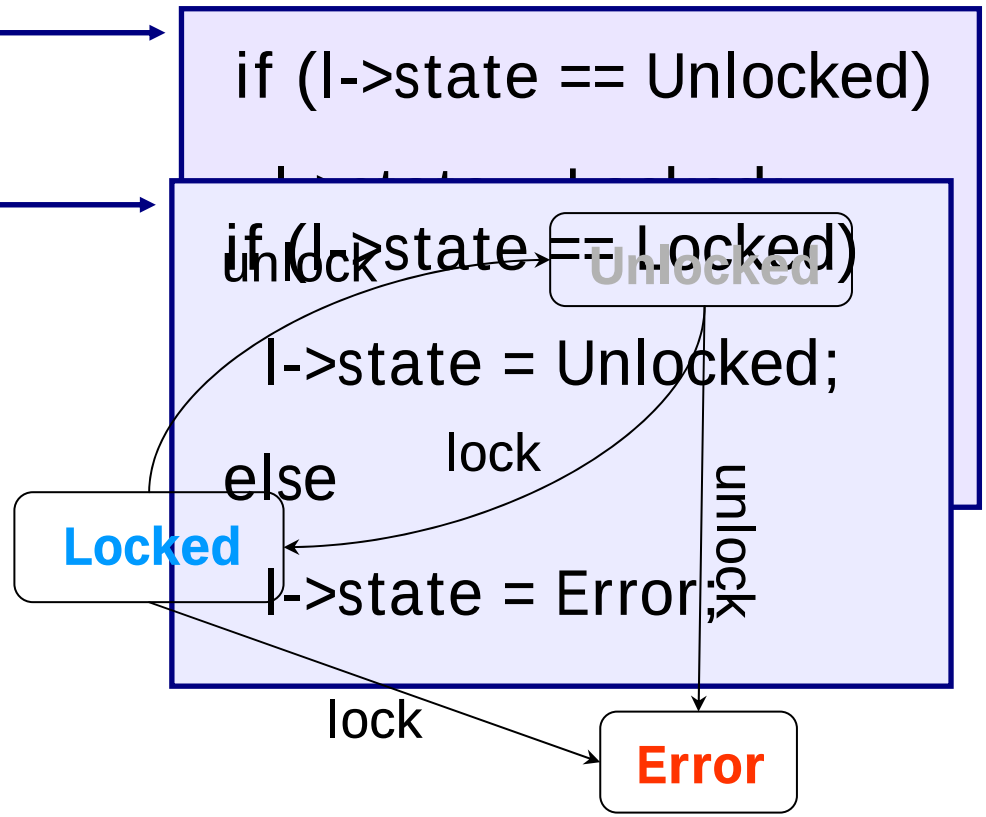
Results for Linux



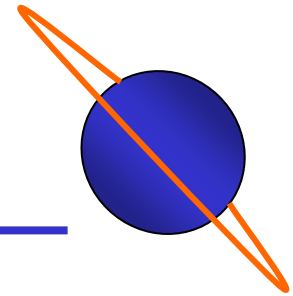
- ~350 bugs
  - And another ~75 false positives (25%)
  - 1 bug per 20,000 lines of code
  - In code already vetted by static analysis tools
- Previous study
  - 52 NULL dereference errors in an earlier Linux
  - < 1 bug per 90,000 lines of code
- Conclusion
  - Scalability & precision matter
  - Many more bugs to be found than have already been found!

Type State: Example Summary Design

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



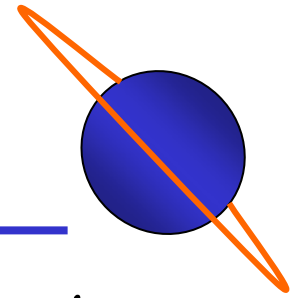
General Type State Checking



- Encode state machine in the program
  - State ➔ Integer
  - Transition ➔ Conditional Assignments

- Check code behavior
  - SAT queries

Function Summaries (1st try)



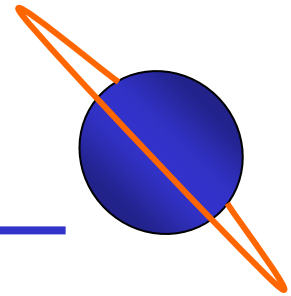
```
int f(lock_t *l)
{
   lock(l);
   …

   …
   unlock(l);
   return 0;
}
```

- Function behavior can be summarized with a set of state transitions

- Summary:

  *l: Unlocked → Unlocked
  Locked → Error

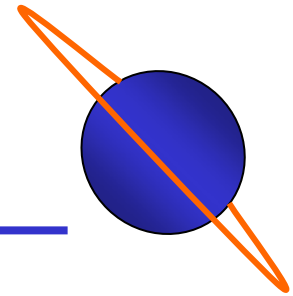
A Difficulty



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

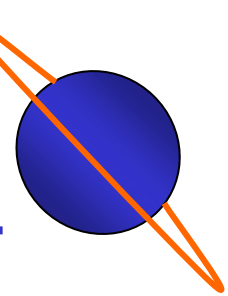
- Problem
  - two possible output states
  - distinguished by return value
    (retval == 0)…
- Summary
  1. (retval == 0)
     *l: Unlocked → Unlocked
     Locked → Error
  2. ¬(retval == 0)
     *l: Unlocked → Locked
     Locked → Error

Type State 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

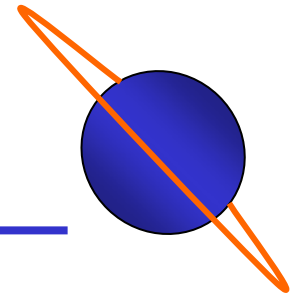
Lock Summary (2nd try)



```
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

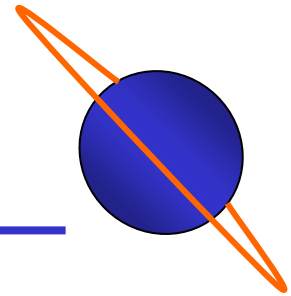
Lock Checker for Linux



- Parameters:
  - States: { Locked, Unlocked, Error }
  - P_{in} = {}
  - P_{out} = { (retval == 0) }

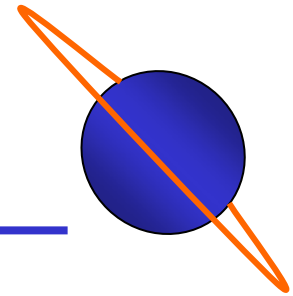
- Experiment:
  - Linux Kernel 2.6.5: 4.8MLOC
  - ~40 lock/unlock/trylock primitives
  - 20 hours to analyze
    - 3.0GHz Pentium IV, 1GB memory

Double Locking/Unlocking



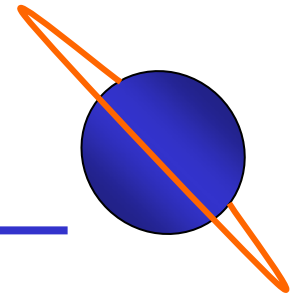
```
static void sscape_coproc_close(…) {
  spin_lock_irqsave(&devc->lock, flags);
  if (…)
      sscape_write(devc, DMAA_REG, 0x20);
  …
}

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

Ambiguous Return State



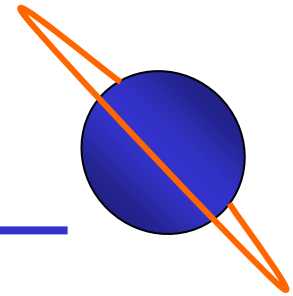
```
int i2o_claim_device(…) {
    down(&i2o_configuration_lock);
    if (d->owner) {
        up(&i2o_configuration_lock);
        return -EBUSY;
    }
    if (…) {
        return -EBUSY;
    }
    …
}
```

Function Summary Database



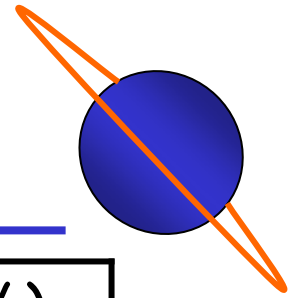
- 63,000 functions in Linux
  - More than 23,000 are lock related
  - 17,000 with locking constraints on entry
  - Around 9,000 affects more than one lock
  - 193 lock wrappers
  - 375 unlock wrappers
  - 36 with return value/lock state correlation

Lock Checker Results on Linux



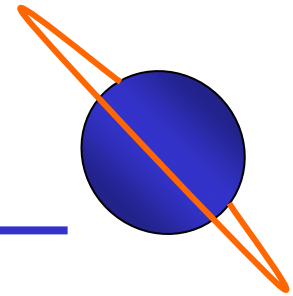
Type	Bugs	False Pos.	% Bugs
Double Locking	134 (31,18)	99	57%(<20%)
Ambiguous State	45	22	67%
Total	179	121	60%

Memory Leak Checker Results



	LOC (K)	# Alloc Func.	# Bugs	FP (%)
Samba	404	80	83	8.79%
OpenSSL	296	101	117	0.85%
BinUtils	909	91	136(66)	3.55%
OpenSSH	36	19	29(10)	0%
Total	1,646	291	365	3.69%

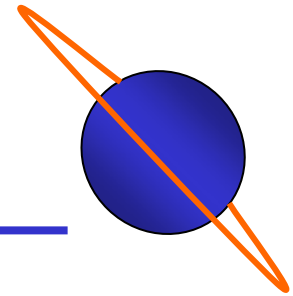
Applications to Verification



- Very much work-in-progress

- One example: user/kernel analysis for Linux
  - Analyzing entire kernel
  - Previous effort:
    - Analyzed 300KLOC
    - Many annotations
    - ~250 false positives

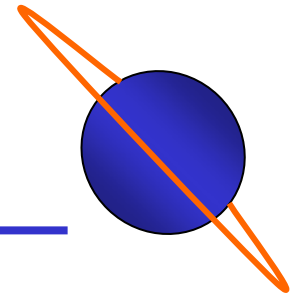
Current and Future Work



- Looking at other applications
  - Null dereference verifier
  - Buffer overruns
  - Integer overflows

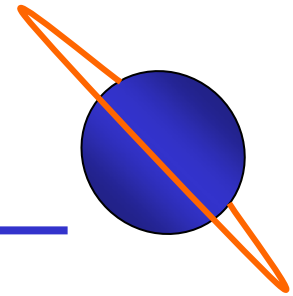
- Using other constraint solvers
  - Linear programming
  - bdd's

Summary



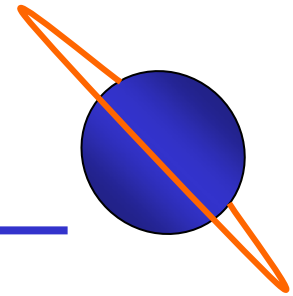
- Need precision within a function
  - Reasoning required is often very complex
  - Often want minimal or no abstraction
  - SAT pays off here
- Across functions, life is simpler
  - Interfaces between functions are much simpler
  - Delay abstraction to function boundaries

Related Work



- Bug Detection Tools
  - SLAM (Ball & Rajamani, MSR)
  - BLAST (Henzinger et. al., UCB)
  - MC (Engler et. al., Stanford)
  - ESP (Das et. al., MSR)
  - PREfix (Bush & Pincus, MSR)
  - CQual (Foster & Aiken, UCB)
- SAT-based tools
  - CBMC (Clarke et. al., CMU)
  - Magic (Chaki et. al., CMU)

The End



- Alex Aiken
- Suhabe Bugrara
- Thomas Dillig
- Brian Hackett
- Peter Hawkins
- Ken Lau
- Nathan Marz
- Isil Ozgener
- Ryan Propper
- Yichen Xie

Saturn release coming in September.