Introduction to Static Analysis

17-654: Analysis of Software Artifacts

Jonathan Aldrich

Find the Bug!

Source: Engler et al., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '00.

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,
                int b_size) {
        struct buffer_head *bh;
        unsigned long flags;

        save_flags(flags);
        cli();
        if ((bh = sh->buffer_pool) == NULL)
                return NULL;
        sh->buffer_pool = bh->b_next;
        bh->b_size = b_size;
        restore_flags(flags);
        return bh;
}
```

disable interrupts

ERROR: returning with interrupts disabled

re-enable interrupts

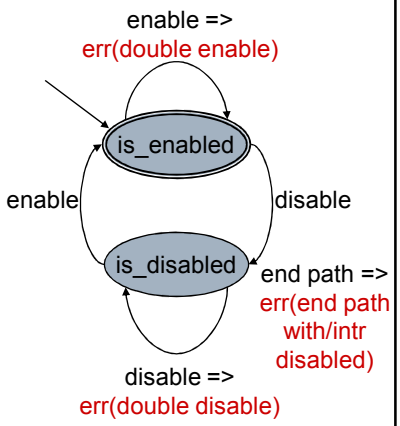
Metal Interrupt Analysis

Source: Engler et al., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '00.

```
{ #include "linux-includes.h" }
sm check_interrupts {
  // Variables
  // used in patterns
  decl { unsigned } flags;

  // Patterns
  // to specify enable/disable functions.
  pat enable = { sti(); }
             | { restore_flags(flags); } ;
  pat disable = { cli(); };

  // States
  // The first state is the initial state.
  is_enabled: disable ==> is_disabled
     | enable ==> { err("double enable"); }
     ;
  is_disabled: enable ==> is_enabled
     | disable ==> { err("double disable"); }
     // Special pattern that matches when the SM
     // hits the end of any path in this state.
     | $end_of_path$ ==>
        { err("exiting w/intr disabled!"); }
     ;
}
```



Applying the Analysis

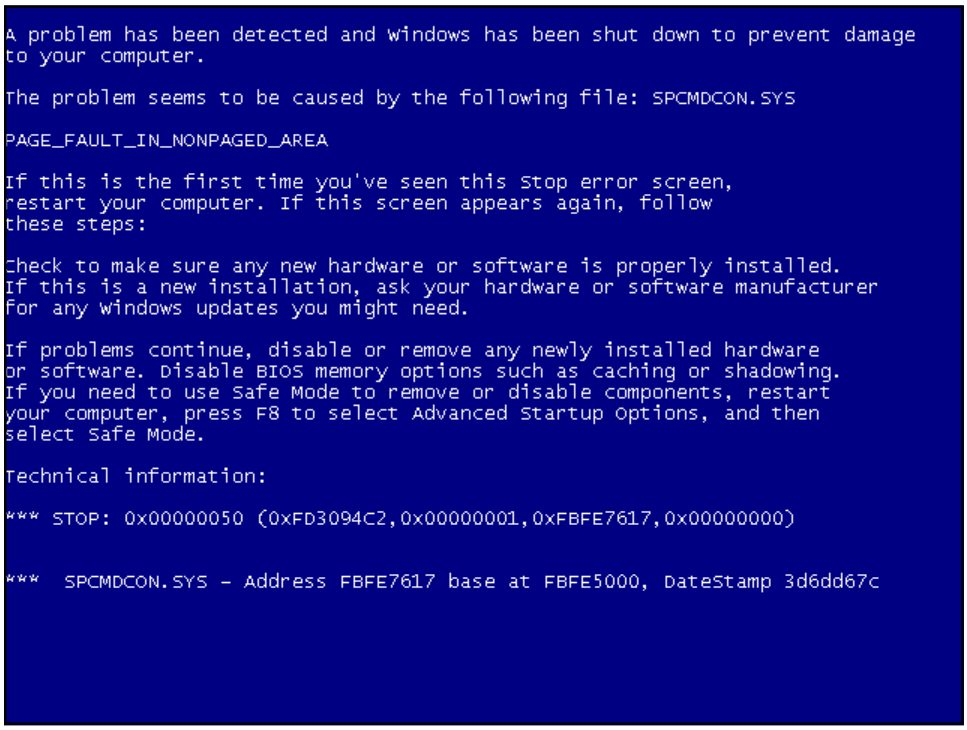
Source: Engler et al., *Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions*, OSDI '00.

```
/* From Linux 2.3.99 drivers/block/raid5.c */
static struct buffer_head *
get_free_buffer(struct stripe_head *sh,        <-- initial state is_enabled
                int b_size) {
        struct buffer_head *bh;
        unsigned long flags;

        save_flags(flags);
        cli();                                 <-- transition to is_disabled
        if ((bh = sh->buffer_pool) == NULL)
                return NULL;                   <-- final state is_disabled: ERROR!
        sh->buffer_pool = bh->b_next;
        bh->b_size = b_size;
        restore_flags(flags);                  <-- transition to is_enabled
        return bh;                             <-- final state is_enabled is OK
}
```

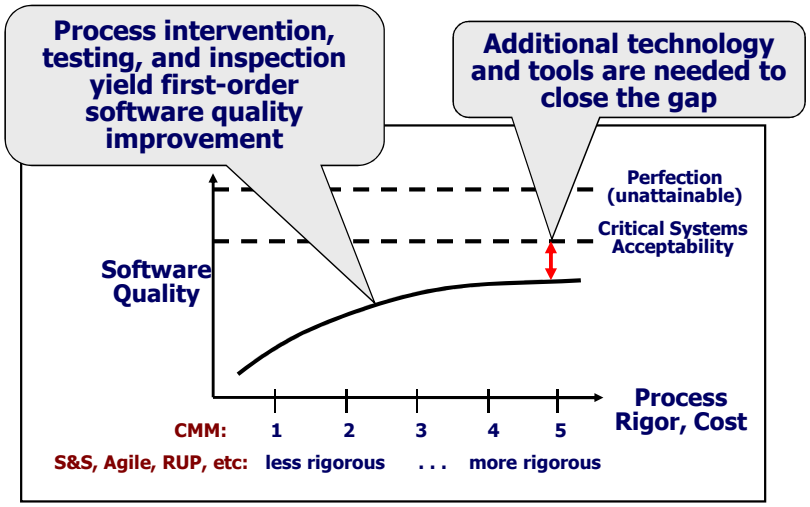
Outline

- Why static analysis?
  - The limits of testing and inspection
- What is static analysis?
- How does static analysis work?
- AST Analysis
- Dataflow Analysis



Process, Cost, and Quality

Slide: William Scherlis



Root Causes of Errors

- Requirements problems
  - Don't fit user needs

Static Analysis Contributions

- Design flaws (Hard)
  - ☆ Lacks required qualities ← Does design achieve goals?
  - ← Is design implemented right?
- Implementation errors
  - Assign ← Is data initialized?
  - Security: Checking ← Is dereference/indexing valid?
  - Algorithm
  - (Hard) ☆ Timing ← Are threads synchronized?
  - (Hard) ☆ Interface ← Are interface semantics followed?
  - (Hard) ☆ Relationship ← Are invariants maintained?

Taxonomy: [Chillarege et al., Orthogonal Defect Classification]

Existing Approaches

- Testing: *is the answer right?*
  - Verifies features work
  - Finds algorithmic problems
- Inspection: *is the quality there?*
  - Missing requirements
  - Design problems
  - Style issues
  - Application logic

- Limitations
  - Non-local interactions
  - Uncommon paths
  - Non-determinism
- Static analysis: *will I get an answer?*
  - Verifies non-local consistency
  - Checks all paths
  - Considers all non-deterministic choices

Static Analysis Finds "Mechanical" Errors

- Defects that result from inconsistently following simple, mechanical design rules

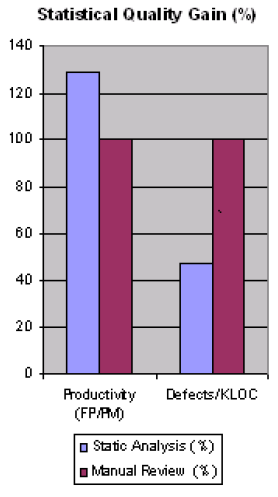
- Security vulnerabilities
  - Buffer overruns, unvalidated input…
- Memory errors
  - Null dereference, uninitialized data…
- Resource leaks
  - Memory, OS resources…
- Violations of API or framework rules
  - e.g. Windows device drivers; real time libraries; GUI frameworks
- Exceptions
  - Arithmetic/library/user-defined
- Encapsulation violations
  - Accessing internal data, calling private functions…
- Race conditions
  - Two threads access the same data without synchronization

Empirical Results on Static Analysis

- Nortel study [Zheng et al. 2006]
  - 3 C/C++ projects
  - 3 million LOC total
  - Early generation static analysis tools
- Conclusions
  - Cost per fault of static analysis 61-72% compared to inspections
  - Effectively finds assignment, checking faults
  - Can be used to find potential security vulnerabilities

Empirical Results on Static Analysis

- InfoSys study [Chaturvedi 2005]
  - 5 projects
  - Average 700 function points each
  - Compare inspection with and without static analysis
- Conclusions
  - Fewer defects
  - Higher productivity



Adapted from [Chaturvedi 2005]

Quality Assurance at Microsoft (Part 1)

- Original process: manual code inspection
  - Effective when system and team are small
  - Too many paths to consider as system grew
- Early 1990s: add massive system and unit testing
  - Tests took weeks to run
    - Diversity of platforms and configurations
    - Sheer volume of tests
  - Inefficient detection of common patterns, security holes
    - Non-local, intermittent, uncommon path bugs
  - Was treading water in Windows Vista development
- Early 2000s: add static analysis
  - More on this later

Outline

- Why static analysis?
- What is static analysis?
  - Abstract state space exploration
- How does static analysis work?
- What do practical tools look like?
- How does it fit into an organization?

Static Analysis Definition

- Static program analysis is the systematic examination of an abstraction of a program's state space
- Metal interrupt analysis
  - Abstraction
    - 2 states: enabled and disabled
      - All program information—variable values, heap contents—is abstracted by these two states, plus the program counter
  - Systematic
    - Examines all paths through a function
      - What about loops? More later…
    - Each path explored for each reachable state
      - Assume interrupts initially enabled (Linux practice)
      - Since the two states abstract all program information, the exploration is exhaustive

Outline

- Why static analysis?
- What is static analysis?
- How does static analysis work?
  - Termination
- AST Analysis
- Dataflow Analysis

How can Analysis Search All Paths?

- How many paths are in a program?
  - Exponential # paths with if statements
  - Infinite # paths with loops
  - How could we possibly cover them all?
- Secret weapon: Abstraction
  - Finite number of (abstract) states
  - If you come to a statement and you've already explored a state for that statement, stop.
    - The analysis depends only on the code and the current state
    - Continuing the analysis from this program point and state would yield the same results you got before
  - If the number of states isn't finite, too bad
    - Your analysis may not terminate

Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.              bar(); cli();
4.      else
5.              baz(); cli();
6.      while (x > 0) {
7.              sti();
8.              do_work();
9.              cli();
10.     }
11.     sti();
12. }
```

Path 1 (before stmt): true/no loop
2: is_enabled
3: is_enabled
6: is_disabled
11: is_disabled
12: is_enabled

no errors

Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.          bar(); cli();
4.      else
5.          baz(); cli();
6.      while (x > 0) {
7.          sti();
8.          do_work();
9.          cli();
10.     }
11.     sti();
12. }
```

Path 2 (before stmt): true/1 loop

2: is_enabled
3: is_enabled
6: is_disabled
7: is_disabled
8: is_enabled
9: is_enabled
11: is_disabled

already been here

Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.          bar(); cli();
4.      else
5.          baz(); cli();
6.      while (x > 0) {
7.          sti();
8.          do_work();
9.          cli();
10.     }
11.     sti();
12. }
```

Path 3 (before stmt): true/2+ loops

2: is_enabled
3: is_enabled
6: is_disabled
7: is_disabled
8: is_enabled
9: is_enabled
6: is_disabled

already been here

Example

```
1.  void foo(int x) {
2.      if (x == 0)
3.          bar(); cli();
4.      else
5.          baz(); cli();
6.      while (x > 0) {
7.          sti();
8.          do_work();
9.          cli();
10.     }
11.     sti();
12. }
```

Path 4 (before stmt): false
2: is_enabled
5: is_enabled
6: is_disabled

already been here

all of state space has been explored

Outline

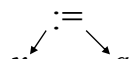
- Why static analysis?
- What is static analysis?
- How does static analysis work?
- AST Analysis
  - Abstract Syntax Tree Representation
  - Simple Bug Finders: FindBugs
- Dataflow Analysis

Representing Programs

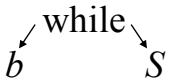
- To analyze software automatically, we must be able to represent it precisely
- Some representations
  - Source code
  - Abstract syntax trees
  - Control flow graph
  - Bytecode
  - Assembly code
  - Binary code

Abstract Syntax Trees

- A tree representation of source code
- Based on the language grammar
  - One type of node for each production
  - $S ::= x := a$ ➔



  - $S ::= \text{while } b \text{ do } S$ ➔



Parsing: Source to AST

- Parsing process (top down)
  1. Determine the top-level production to use
  2. Create an AST element for that production
  3. Determine what text corresponds to each child of the AST element
  4. Recursively parse each child
- Algorithms have been studied in detail
  - For this course you only need the intuition
  - Details covered in compiler courses

Parsing Example

```
y := x;
z := 1;
while y>1 do
    z := z * y;
    y := y – 1
```

- Top-level production?
- What are the parts?

Parsing Example

y := x; ;

z := 1;

while y>1 do

    z := z * y;

    y := y – 1

- Top-level production?
  - $S_1; S_2$
- What are the parts?

Parsing Example

y := x;

z := 1;

while y>1 do

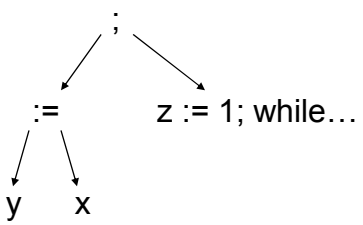
    z := z * y;

    y := y – 1

- Top-level production?
  - $S_1; S_2$
- What are the parts?
  - y := x
  - z := 1; while …

Parsing Example

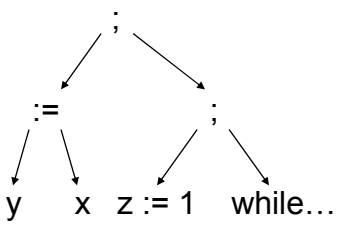
```
y := x;
z := 1;
while y>1 do
    z := z * y;
    y := y – 1
```



- Top-level production?
  - $S_1; S_2$
- What are the parts?
  - y := x
  - z := 1; while …

Parsing Example

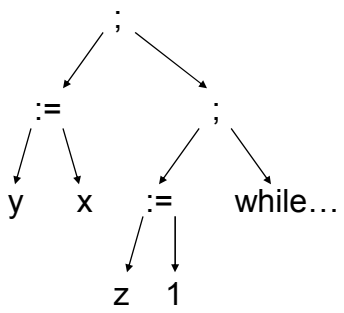
```
y := x;
z := 1;
while y>1 do
    z := z * y;
    y := y – 1
```



- Top-level production?
  - $S_1; S_2$
- What are the parts?
  - y := x
  - z := 1; while …

Parsing Example

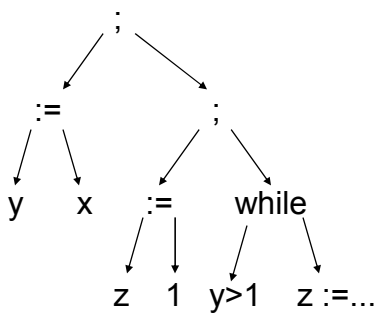
```
y := x;
z := 1;
while y>1 do
    z := z * y;
    y := y – 1
```



- Top-level production?
  - $S_1; S_2$
- What are the parts?
  - y := x
  - z := 1; while …

Parsing Example

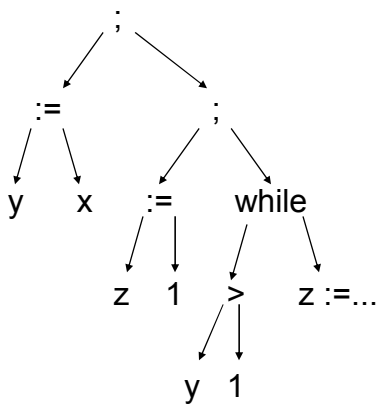
```
y := x;
z := 1;
while y>1 do
    z := z * y;
    y := y – 1
```



- Top-level production?
  - $S_1; S_2$
- What are the parts?
  - y := x
  - z := 1; while …

Parsing Example

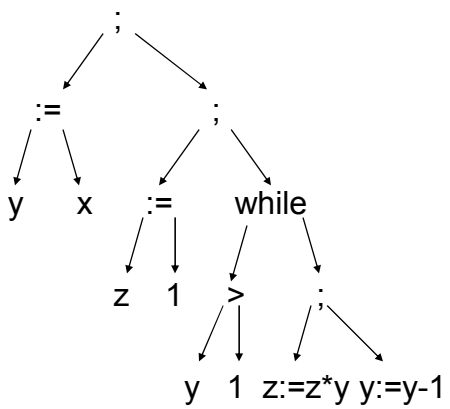
```
y := x;
z := 1;
while y>1 do
    z := z * y;
    y := y – 1
```



- Top-level production?
  - $S_1; S_2$
- What are the parts?
  - y := x
  - z := 1; while …

Parsing Example

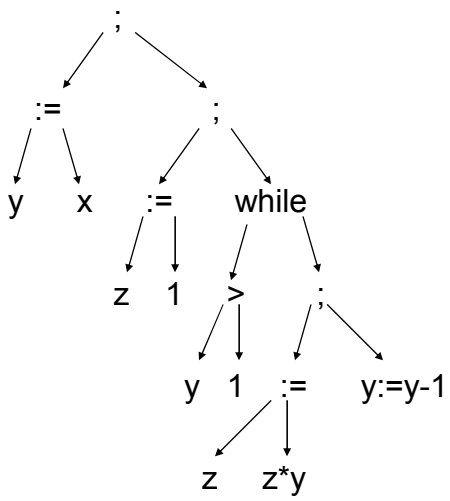
```
y := x;
z := 1;
while y>1 do
    z := z * y;
    y := y – 1
```



- Top-level production?
  - $S_1; S_2$
- What are the parts?
  - y := x
  - z := 1; while …

Parsing Example

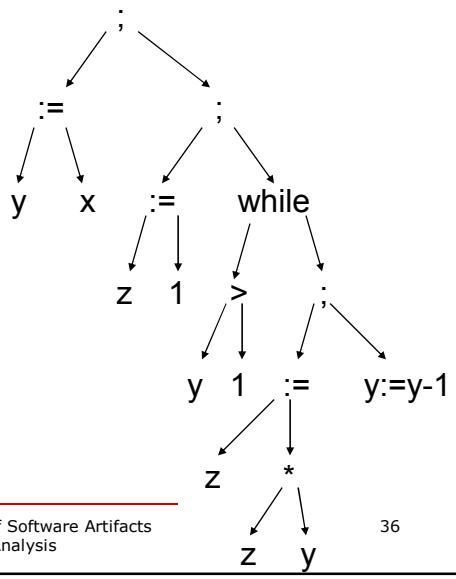
```
y := x;
z := 1;
while y>1 do
    z := z * y;
    y := y – 1
```



- Top-level production?
  - $S_1; S_2$
- What are the parts?
  - y := x
  - z := 1; while …

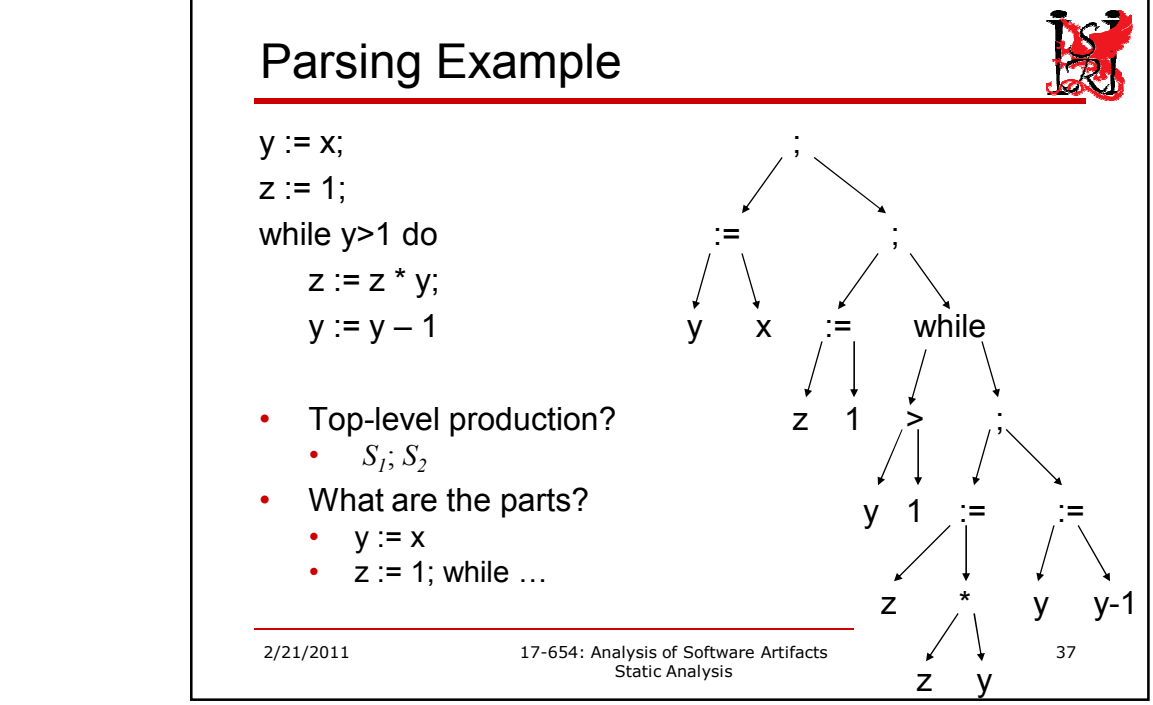
Parsing Example

```
y := x;
z := 1;
while y>1 do
    z := z * y;
    y := y – 1
```



- Top-level production?
  - $S_1; S_2$
- What are the parts?
  - y := x
  - z := 1; while …

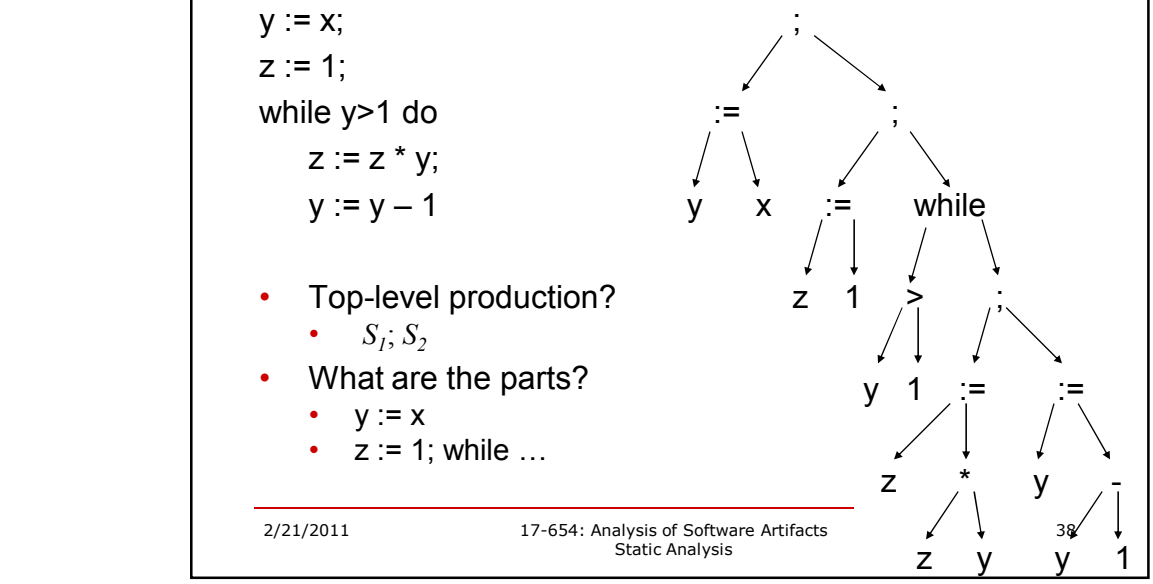
Parsing Example



```
y := x;
z := 1;
while y>1 do
    z := z * y;
    y := y – 1
```

- Top-level production?
  - $S_1; S_2$
- What are the parts?
  - y := x
  - z := 1; while …

Parsing Example



```
y := x;
z := 1;
while y>1 do
    z := z * y;
    y := y – 1
```

- Top-level production?
  - $S_1; S_2$
- What are the parts?
  - y := x
  - z := 1; while …

Quick Quiz

Draw a parse tree for the function below. You can assume that the "for" statement is at the top of the parse tree.

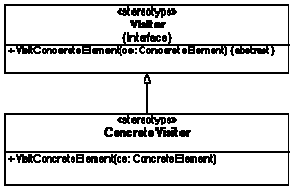
```
void copy_bytes(char dest[], char source[], int n) {
    for (int i = 0; i < n; ++i)
        dest[i] = source[i];
}
```

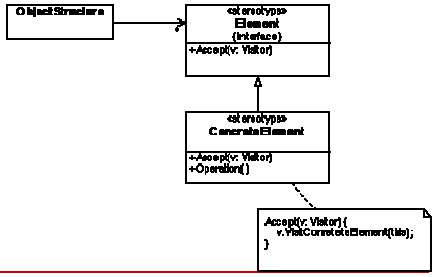

Matching AST against Bug Patterns

- AST Walker Analysis
  - Walk the AST, looking for nodes of a particular type
  - Check the immediate neighborhood of the node for a bug pattern
  - Warn if the node matches the pattern
- Semantic grep
  - Like grep, looking for simple patterns
  - Unlike grep, consider not just names, but semantic structure of AST
    - Makes the analysis more precise
- Common architecture based on Visitors
  - class Visitor has a visitX method for each type of AST node X
  - Default Visitor code just descends the AST, visiting each node
  - To find a bug in AST element of type X, override visitX

Behavioral Patterns: Visitor

- Applicability
  - Structure with many classes
  - Want to perform operations that depend on classes
  - Set of classes is stable
  - Want to define new operations
- Consequences
  - Easy to add new operations
  - Groups related behavior in Visitor
  - Adding new elements is hard
  - Visitor can store state
  - Elements must expose interface





Example: Shifting by more than 31 bits

```
class BadShiftAnalysis extends Visitor
    visitShiftExpression(ShiftExpression e) {
        if (type of e's left operand is int)
            if (e's right operand is a constant)
                if (value of constant < 0 or > 31)
                        warn("Shifting by less than 0 or more
                              than 31 is meaningless")

        super.visitShiftExpression(e);
    }
```

Example: String concatenation in a loop

```
class StringConcatLoopAnalysis extends Visitor
    private int loopLevel = 0;

    visitStringConcat(StringConcat e) {
        if (loopLevel > 0)
            warn("Performance issue: String concatenation in loop (use
    StringBuffer instead)")
        super.visitStringConcat(e);      // visits AST children
    }

    visitWhile(While e) {
        loopLevel++;
        super.visitWhile(e);             // visits AST children
        loopLevel--;
    }
    // similar for other looping constructs
```


Example Tool: FindBugs

- Origin: research project at U. Maryland
  - Now freely available as open source
  - Standalone tool, plugins for Eclipse, etc.
- Checks over 250 "bug patterns"
  - Over 100 correctness bugs
  - Many style issues as well
  - Includes the two examples just shown
- Focus on simple, local checks
  - Similar to the patterns we've seen
  - But checks bytecode, not AST
    - Harder to write, but more efficient and doesn't require source
- http://findbugs.sourceforge.net/

Example FindBugs Bug Patterns

- Correct equals()
- Use of ==
- Closing streams
- Illegal casts
- Null pointer dereference
- Infinite loops
- Encapsulation problems
- Inconsistent synchronization
- Inefficient String use
- Dead store to variable

FindBugs Experiences

- Useful for learning idioms of Java
  - Rules about libraries and interfaces
    - e.g. equals()
- Customization is important
  - Many warnings may be irrelevant, others may be important – depends on domain
    - e.g. embedded system vs. web application
- Useful for pointing out things to examine
  - Not all are real defects
  - Turn off false positive warnings for future analyses on codebase

Outline

- Why static analysis?
- What is static analysis?
- How does static analysis work?
- AST Analysis
- Dataflow Analysis
  - Control Flow Graph Representation
  - Simple Flow Analysis: Zero/Null Values

Motivation: Dataflow Analysis

- Catch interesting errors
  - Non-local: x is null, x is written to y, y is dereferenced
- Optimize code
  - Reduce run time, memory usage…
- Soundness required
  - Safety-critical domain
    - Assure lack of certain errors
  - Cannot optimize unless it is proven safe
    - Correctness comes before performance
- Automation required
  - Dramatically decreases cost
  - Makes cost/benefit worthwhile for far more purposes

Dataflow analysis

- Tracks value flow through program
  - Can distinguish order of operations
    - Did you read the file after you closed it?
    - Does this null value flow to that dereference?
  - Differs from AST walker
    - Walker simply collects information or checks patterns
    - Tracking flow allows more interesting properties
- Abstracts values
  - Chooses abstraction particular to property
    - Is a variable null?
    - Is a file open or closed?
    - Could a variable be 0?
    - Where did this value come from?
  - More *specialized* than Hoare logic
    - Hoare logic allows any property to be expressed
    - Specialization allows automation and soundness

Zero Analysis

- Could variable x be 0?
  - Useful to know if you have an expression y/x
  - In C, useful for null pointer analysis
- Program semantics
  - η maps every variable to an integer
- Semantic abstraction
  - σ maps every variable to non zero (NZ), zero(Z), or maybe zero (MZ)
  - Abstraction function for integers α_{ZI} :
    - $\alpha_{ZI}(0)$ = Z
    - $\alpha_{ZI}(n)$ = NZ for all $n \neq 0$
  - We may not know if a value is zero or not
    - Analysis is always an approximation
    - Need MZ option, too

Zero Analysis Example

σ =[]

```
x := 10;
y := x;
z := 0;
while y > -1 do
    x := x / y;
    y := y-1;
    z := 5;
```


Zero Analysis Example

σ =[]

```
x := 10;
```

σ =[x↦α_{ZI}(10)]

```
y := x;
z := 0;
while y > -1 do
    x := x / y;
    y := y-1;
    z := 5;
```

Zero Analysis Example

```
                              σ=[]
x := 10;                      σ=[x↦NZ]
y := x;
z := 0;
while y > -1 do
    x := x / y;
    y := y-1;
    z := 5;
```


Zero Analysis Example

```
                              σ=[]
x := 10;                      σ=[x↦NZ]
y := x;                       σ=[x↦NZ,y↦σ(x)]
z := 0;
while y > -1 do
    x := x / y;
    y := y-1;
    z := 5;
```

Zero Analysis Example

```
x := 10;
y := x;
z := 0;
while y > -1 do
    x := x / y;
    y := y-1;
    z := 5;
```

σ =[]
σ =[x↦NZ]
σ =[x↦NZ,y↦NZ]

Zero Analysis Example

```
x := 10;
y := x;
z := 0;
while y > -1 do
    x := x / y;
    y := y-1;
    z := 5;
```

σ =[]
σ =[x↦NZ]
σ =[x↦NZ,y↦NZ]
σ =[x↦NZ,y↦NZ,z↦α_{ZI}(0)]

Zero Analysis Example

	σ =[]
x := 10;	σ =[x↦NZ]
y := x;	σ =[x↦NZ,y↦NZ]
z := 0;	σ =[x↦NZ,y↦NZ,z↦Z]
while y > -1 do	
x := x / y;	
y := y-1;	
z := 5;	

Zero Analysis Example

	σ =[]
x := 10;	σ =[x↦NZ]
y := x;	σ =[x↦NZ,y↦NZ]
z := 0;	σ =[x↦NZ,y↦NZ,z↦Z]
while y > -1 do	σ =[x↦NZ,y↦NZ,z↦Z]
x := x / y;	
y := y-1;	
z := 5;	

Zero Analysis Example

	σ =[]
x := 10;	σ =[x↦NZ]
y := x;	σ =[x↦NZ,y↦NZ]
z := 0;	σ =[x↦NZ,y↦NZ,z↦Z]
while y > -1 do	σ =[x↦NZ,y↦NZ,z↦Z]
x := x / y;	σ =[x↦NZ,y↦NZ,z↦Z]
y := y-1;	
z := 5;	

Zero Analysis Example

	σ =[]
x := 10;	σ =[x↦NZ]
y := x;	σ =[x↦NZ,y↦NZ]
z := 0;	σ =[x↦NZ,y↦NZ,z↦Z]
while y > -1 do	σ =[x↦NZ,y↦NZ,z↦Z]
x := x / y;	σ =[x↦NZ,y↦NZ,z↦Z]
y := y-1;	σ =[x↦NZ,y↦MZ,z↦Z]
z := 5;	

Zero Analysis Example

	σ =[]
x := 10;	σ =[x↦NZ]
y := x;	σ =[x↦NZ,y↦NZ]
z := 0;	σ =[x↦NZ,y↦NZ,z↦Z]
while y > -1 do	σ =[x↦NZ,y↦NZ,z↦Z]
x := x / y;	σ =[x↦NZ,y↦NZ,z↦Z]
y := y-1;	σ =[x↦NZ,y↦MZ,z↦Z]
z := 5;	σ =[x↦NZ,y↦MZ,z↦NZ]

Zero Analysis Example

	σ =[]
x := 10;	σ =[x↦NZ]
y := x;	σ =[x↦NZ,y↦NZ]
z := 0;	σ =[x↦NZ,y↦NZ,z↦Z]
while y > -1 do	σ =[x↦NZ,y↦MZ,z↦MZ]
x := x / y;	σ =[x↦NZ,y↦NZ,z↦Z]
y := y-1;	σ =[x↦NZ,y↦MZ,z↦Z]
z := 5;	σ =[x↦NZ,y↦MZ,z↦NZ]

Zero Analysis Example

	σ =[]
x := 10;	σ =[x↦NZ]
y := x;	σ =[x↦NZ,y↦NZ]
z := 0;	σ =[x↦NZ,y↦NZ,z↦Z]
while y > -1 do	σ =[x↦NZ,y↦MZ,z↦MZ]
x := x / y;	σ =[x↦NZ,y↦MZ,z↦MZ]
y := y-1;	σ =[x↦NZ,y↦MZ,z↦Z]
z := 5;	σ =[x↦NZ,y↦MZ,z↦NZ]

Zero Analysis Example

	σ =[]
x := 10;	σ =[x↦NZ]
y := x;	σ =[x↦NZ,y↦NZ]
z := 0;	σ =[x↦NZ,y↦NZ,z↦Z]
while y > -1 do	σ =[x↦NZ,y↦MZ,z↦MZ]
x := x / y;	σ =[x↦NZ,y↦MZ,z↦MZ]
y := y-1;	σ =[x↦NZ,y↦MZ,z↦MZ]
z := 5;	σ =[x↦NZ,y↦MZ,z↦NZ]

Zero Analysis Example

	σ=[]
x := 10;	σ=[x↦NZ]
y := x;	σ=[x↦NZ,y↦NZ]
z := 0;	σ=[x↦NZ,y↦NZ,z↦Z]
while y > -1 do	σ=[x↦NZ,y↦MZ,z↦MZ]
x := x / y;	σ=[x↦NZ,y↦MZ,z↦MZ]
y := y-1;	σ=[x↦NZ,y↦MZ,z↦MZ]
z := 5;	σ=[x↦NZ,y↦MZ,z↦NZ]
	Nothing more happens!

Zero Analysis Termination

- The analysis values will not change, no matter how many times we execute the loop
  - Proof: our analysis is deterministic
  - We run through the loop with the current analysis values, none of them change
  - Therefore, no matter how many times we run the loop, the results will remain the same
  - Therefore, we have computed the dataflow analysis results for any number of loop iterations

Zero Analysis Termination

- The analysis values will not change, no matter how many times we execute the loop
  - Proof: our analysis is deterministic
  - We run through the loop with the current analysis values, none of them change
  - Therefore, no matter how many times we run the loop, the results will remain the same
  - Therefore, we have computed the dataflow analysis results for any number of loop iterations
- Why does this work
  - If we simulate the loop, the data values could (in principle) keep changing indefinitely
    - There are an infinite number of data values possible
    - Not true for 32-bit integers, but might as well be true
      - Counting to 2^{32} is slow, even on today's processors
  - Dataflow analysis only tracks 2 possibilities!
    - So once we've explored them all, nothing more will change
    - This is the secret of abstraction
- We will make this argument more precise later

Using Zero Analysis

- Visit each division in the program
- Get the results of zero analysis for the divisor
- If the results are definitely zero, report an error
- If the results are possibly zero, report a warning

Quick Quiz

- Fill in the table to show how what information zero analysis will compute for the function given.

Program Statement	Analysis Info after that statement
0: <*beginning of program*>	
1: x := 0	
2: y := 1	
3: if (z == 0)	
4: x := x + y	
5: else y := y – 1	
6: w := y	

Outline

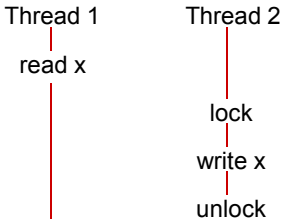
- Why static analysis?
- What is static analysis?
- How does static analysis work?
- AST Analysis
- Dataflow Analysis
- Further Examples and Discussion

Static Analysis Definition

- Static program analysis is the systematic examination of an abstraction of a program's state space
- Simple model checking for data races
  - ***Data Race*** defined:
    [From Savage et al., *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*]
    - Two threads access the same variable
    - At least one access is a write
    - No explicit mechanism prevents the accesses from being simultaneous
  - Abstraction
    - Program counter of each thread, state of each lock
      - Abstract away heap and program variables
  - Systematic
    - Examine all possible interleavings of all threads
      - Flag error if no synchronization between accesses
      - Exploration is exhaustive, since abstract state abstracts all concrete program state

Model Checking for Data Races

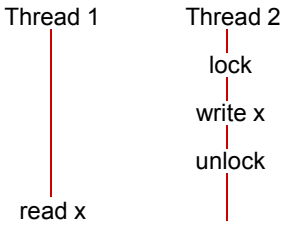
```
thread1() {
   read x;
}
thread2() {
   lock();
   write x;
   unlock();
}
```



Interleaving 1: OK

Model Checking for Data Races

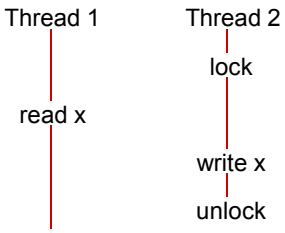
```
thread1() {
   read x;
}
thread2() {
   lock();
   write x;
   unlock();
}
```



Interleaving 1: OK
Interleaving 2: OK

Model Checking for Data Races

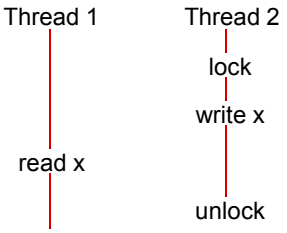
```
thread1() {
   read x;
}
thread2() {
   lock();
   write x;
   unlock();
}
```



Interleaving 1: OK
Interleaving 2: OK
Interleaving 3: Race

Model Checking for Data Races

```
thread1() {
   read x;
}
thread2() {
   lock();
   write x;
   unlock();
}
```



Interleaving 1: OK
Interleaving 2: OK
Interleaving 3: Race
Interleaving 4: Race

Compare Analysis to Testing, Inspection

- Why might it be hard to test/inspect for:
  - Null pointer errors?
  - Forgetting to re-enable interrupts?
  - Race conditions?

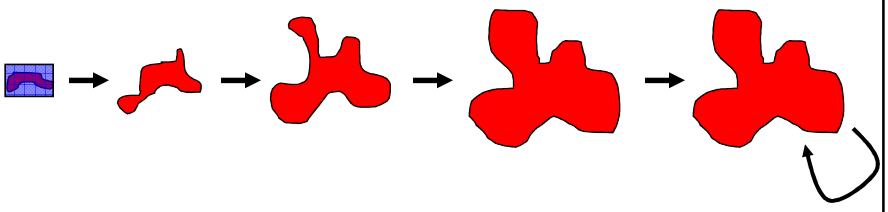
Compare Analysis to Testing, Inspection

- Null Pointers, Interrupts
  - Testing
    - Errors typically on uncommon paths or uncommon input
    - Difficult to exercise these paths
  - Inspection
    - Non-local and thus easy to miss
      - Object allocation vs. dereference
      - Disable interrupts vs. return statement
- Finding Data Races
  - Testing
    - Cannot force all interleavings
  - Inspection
    - Too many interleavings to consider
    - Check rules like “lock protects x” instead
      - But checking is non-local and thus easy to miss a case

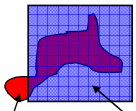
Sound Analyses

- A sound analysis never misses an error
  [of the relevant error category]
  - No *false negatives (missed errors)*
  - Requires exhaustive exploration of state space
- Inductive argument for soundness
  - Start program with abstract state for all possible initial concrete states
  - At each step, ensure new abstract state covers all concrete states that could result from executing statement on any concrete state from previous abstract state
  - Once no new abstract states are reachable, by induction all concrete program executions have been considered

Soundness and Precision



Program state covered in actual execution

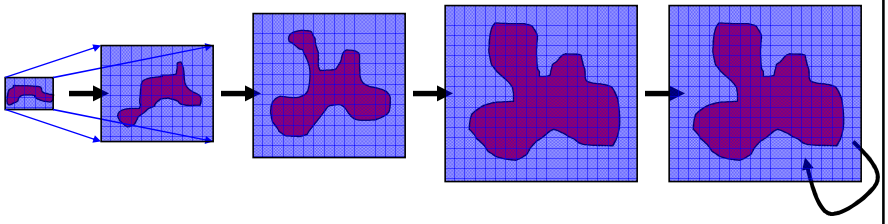


unsound (false negative)

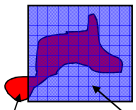
imprecise (false positive)

Program state covered by abstract execution with analysis

Soundness and Precision



Program state covered in actual execution



unsound (false negative)

imprecise (false positive)

Program state covered by abstract execution with analysis

Abstraction and Soundness

- Consider "Sound Testing"
  [testing that finds every bug]
  - Requires executing program on every input
    - (and on all interleavings of threads)
  - Infinite number of inputs for realistic programs
    - Therefore impossible in practice
- Abstraction
  - Infinite state space → finite set of states
  - Can achieve soundness by exhaustive exploration

Zero Analysis Precision

```
1.  void foo(unsigned n) {
2.      int x = -1;
3.      x = x+2;
4.      int y = 10/x;
5.  }
```

What will be the result of static analysis?

Path 1 (after stmt):
1: ∅
2: x↦NZ
3: x↦MZ

warning: possible divide by zero at line 4
False positive! (not a real error)

What went wrong?
- Before statement 3 we only know x is nonzero
- We need to know that x is -1

Regaining Zero Analysis Precision

- Keep track of exact value of variables
  - Infinite states
    - or 2^{32}, close enough
- Add a -1 state
  - Not general enough
- Track formula for every variable
  - Undecidable for arbitrary formulas
- Track restricted formulas
  - Decent solution in practice
    - Presburger arithmetic

Analysis as an Approximation

- Analysis must approximate in practice
  - May report errors where there are really none
    - False positives
  - May not report errors that really exist
    - False negatives
  - All analysis tools have either false negatives or false positives
- Approximation strategy
  - Find a pattern P for correct code
    - which is feasible to check (analysis terminates quickly),
    - covers most correct code in practice (low false positives),
    - which implies no errors (no false negatives)
- Analysis can be pretty good in practice
  - Many tools have low false positive/negative rates
  - A sound tool has no false negatives
    - Never misses an error in a category that it checks

Attribute-Specific Analysis

- Analysis is specific to
  - A quality attribute
    - Race condition
    - Buffer overflow, divide by zero
    - Use after free
  - A strategy for verifying that attribute
    - Protect each shared piece of data with a lock
    - Presburger arithmetic decision procedure for array indexes, zero analysis
    - Only one variable points to each memory location
- Analysis is inappropriate for some attributes
  - Approach to assurance is ad-hoc and follows no clear pattern
  - No known decision procedure for checking an assurance pattern that is followed
  - ***Examples?***

Soundness Tradeoffs

- Sound Analysis
  - Assurance that no bugs are left
    - Of the target error class
  - Can focus other QA resources on other errors
  - May have more false positives
- Unsound Analysis
  - No assurance that bugs are gone
  - Must still apply other QA techniques
  - May have fewer false positives

Which to Choose?

- Cost/Benefit tradeoff
  - Benefit: How valuable is the bug?
    - How much does it cost if not found?
    - How expensive to find using testing/inspection?
  - Cost: How much did the analysis cost?
    - Effort spent running analysis, interpreting results – includes false positives
    - Effort spent finding remaining bugs (for unsound analysis)
- Rule of thumb
  - For critical bugs that testing/inspection can't find, a sound analysis is worth it
    - As long as false positive rate is acceptable
  - For other bugs, maximize engineer productivity

Questions?

Additional Slides/Examples

Static Analysis Definition

- Static program analysis is the systematic examination of an abstraction of a program's state space
- Simple array bounds analysis
  - Abstraction
    - Given array *a*, track whether each integer variable and expression is <,=, or > than *length(a)*
      - Abstract away precise values of variables and expressions
      - Abstract away the heap
  - Systematic
    - Examines all paths through a function
    - Each path explored for each reachable state
      - Exploration is exhaustive, since abstract state abstracts all concrete program state

Array Bounds Example

```
1.  void foo(unsigned n) {
2.      char str = new char[n+1];
3.      int idx = 0;
4.      if (n > 5)
5.              idx = n
6.      else
7.              idx = n+1
8.      str[idx] = 'c';
9.  }
```

Path 1 (before stmt): then branch

2: ∅

3: n↦<

4: n↦<, idx↦<

5: n↦<, idx↦<

8: n↦<, idx↦<

9: n↦<, idx↦<

no errors

Array Bounds Example

```
1.  void foo(unsigned n) {
2.      char str = new char[n+1];
3.      int idx = 0;
4.      if (n > 5)
5.              idx = n
6.      else
7.              idx = n+1
8.      str[idx] = 'c';
9.  }
```

Path 1 (before stmt): else branch

2: ∅

3: n↦<

4: n↦<, idx↦<

7: n↦<, idx↦<,=

8: n↦<, idx↦<,=

9: n↦<, idx↦<,=

error: array out of bounds at line 8