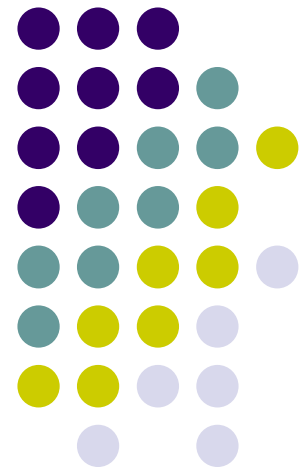


# Static Code Analysis

Lecture 8  
Oct 5, 2017

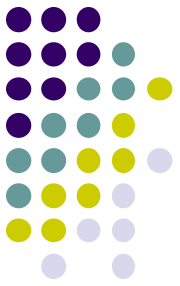


Source:  
“*Secure Programming with Static Analysis*”



# Static Analysis

- Analyzing code before executing it
  - Analogy: Spell checker
- Suited to problem identification because
  - Checks thoroughly and consistently
  - Can point to the root cause of the problem
    - E.g., presence of buffer overflow; helps to focus on what to fix
  - Help find errors/bugs early in the development
    - Helps reduce cost
  - New information can be easily incorporated to recheck a given program



# Usefulness

- Better than manual code review
- Faster and more concrete than testing
- Consistency in coverage
- Embody the existing security knowledge and gets extended
- Great for use by non-experts

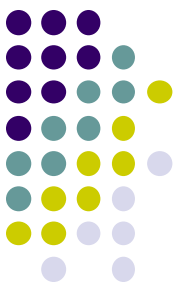


# Key Issues

- Can give a lot of noise!
- False Positives & False Negative
  - Which is worse? Need to balance the FP and FN
- Defects must be visible to the tool
- Different types of Static analysis:
  - Type checking; Style checking
  - Program understanding ; Program verification
  - Property checking; Bug finding
  - Security Review

***It is Computationally undecidable problem***

# Type Checking



**Example 2.1** A type-checking false positive: These Java statements do not meet type safety rules even though they are logically correct.

```
10 short s = 0;
11 int i = s; /* the type checker allows this */
12 short r = i; /* false positive: this will cause a
13              type checking error at compile time */
```



**Example 2.2** Output from the Java compiler demonstrating the type-checking false positive.

```
$ javac bar.java
bar.java:12: possible loss of precision
found   : int
required: short
    short r = i; /* false positive: this will cause a
                ^
1 error
```

**Example 2.3** These Java statements meet type-checking rules but will fail at runtime.

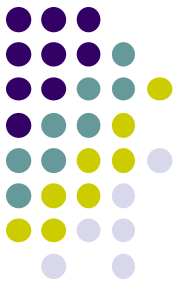
```
Object[] objs = new String[1];
objs[0] = new Object();
```





# Style Checking

- Superficial set of rules
- Focused on rules related to
  - Whitespace, naming, deprecated functions, commenting, program structure
  - Affect: [readability](#) and maintainability rather than coding error
  - [-Wall](#) in [gcc](#)
    - Detect when a switch statement does not account for all possible values
  - For a large project many people with their own style may be involved
  - Examples: [lint](#), [PMD](#)



# Program Understanding

- Helps make sense of a large Codebase
  - Examples
    - Tool example: Fujaba
      - UML and Java Code – can help back and forth
      - “Finding all uses of a method”
      - “Finding declaration of a global variable”
  - Helpful to work on code one has not written
    - some reverse engineer the design – “big picture”
  - IDEs typically include some PU functionality

# Program verification and Property checking



## Memory leak

- Accepts a specification and associated Code

- Aims to prove that the code is faithful implementation
- “*equivalence checking*” to check the two match

```
1  inBuf = (char*) malloc(bufSz);
2  if (inBuf == NULL)
3      return -1;
4  outBuf = (char*) malloc(bufSz);
5  if (outBuf == NULL)
6      return -1;
```

- Complete specific consuming !

Violation of property "allocated memory should always be freed":

line 2: inBuf != NULL

line 5: outBuf == NULL

line 6: function returns (-1) without freeing inBuf

- So “Partial” verification

- Try to find a “cc

- Sound wrt the spec

- It will always return a problem if one exists !
  - (false negative? False positive?)
  - Soundness may be very difficult to establish

**Counter example for:  
Allocated memory  
should always be  
freed**





# Bug Finding

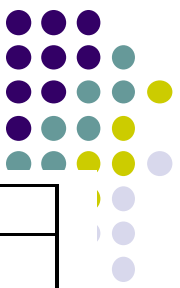
- Points out places where the program will behave in a way that the coder did not intend
  - Use patterns that indicate bugs
  - Example: FindBug (Java), Coverity (C, C++)
- Early tools: ITS4, RATS, Flawfinder
  - Little more than glorified “grep”
  - Closer to style checkers
- Modern tools
  - Typically hybrid of property checkers and bug finders



# Factors for utility of SA

- Ability of the tool to make sense of the program
- Trade-offs it makes between precision and scalability
- Errors that it can check/detect
- How easily usable by programmers/users

# Some examples



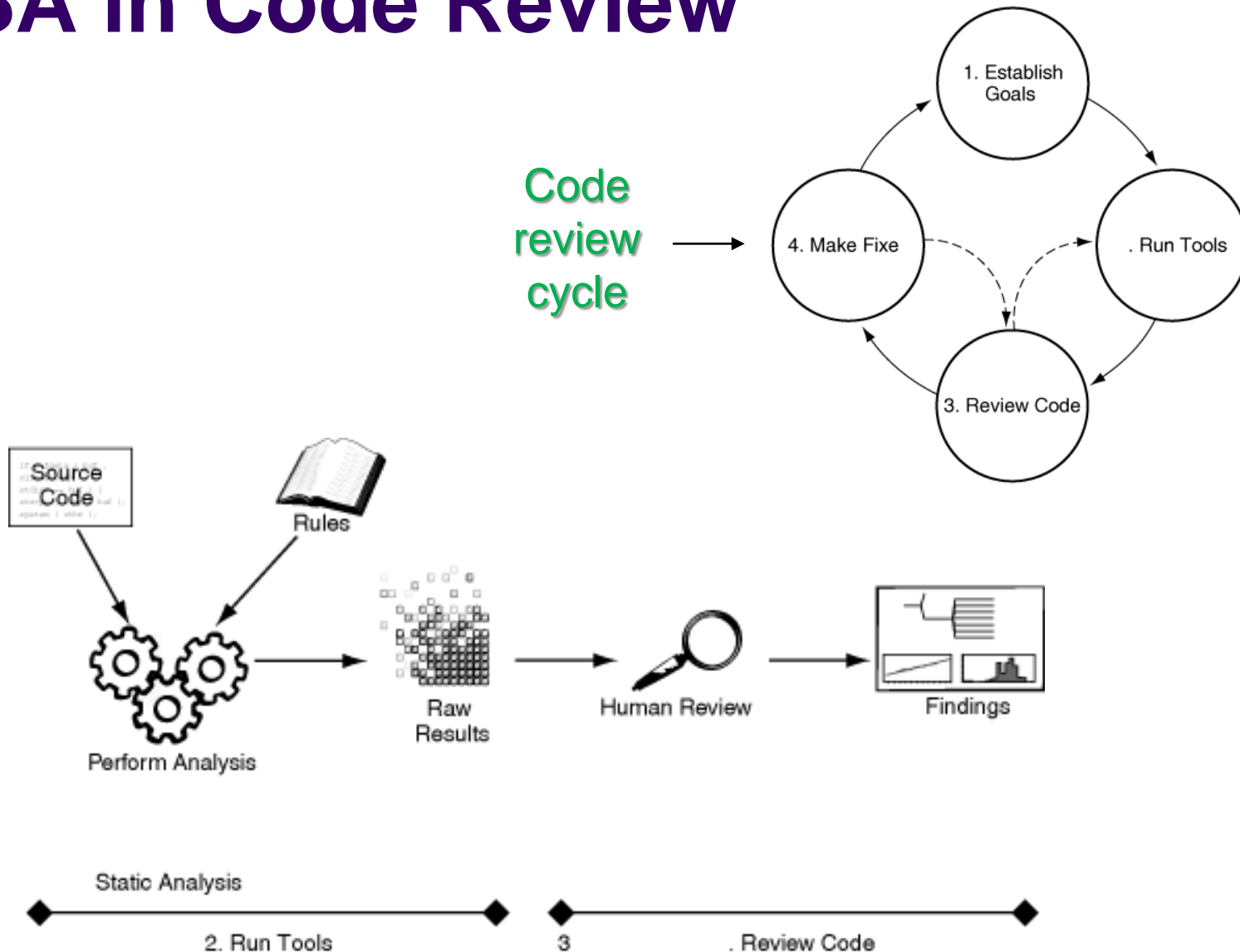
Type of Tool/Vendors	Web Site
<u>Style Checking</u>	
PMD	<a href="http://pmd.sourceforge.net">http://pmd.sourceforge.net</a>
Parasoft	<a href="http://www.parasoft.com">http://www.parasoft.com</a>
<u>Program Understanding</u>	
Fujaba	<a href="http://www.wcs.uni-paderborn.de/cs/fujaba/">http://www.wcs.uni-paderborn.de/cs/fujaba/</a>
CAST	<a href="http://www.castsoftware.com">http://www.castsoftware.com</a>
<u>Program Verification</u>	
Praxis High Integrity Systems	<a href="http://www.praxis-his.com">http://www.praxis-his.com</a>
Escher Technologies	<a href="http://www.eschertech.com">http://www.eschertech.com</a>
<u>Property Checking</u>	
Polyspace	<a href="http://www.polyspace.com">http://www.polyspace.com</a>
Grammatech	<a href="http://www.grammatech.com">http://www.grammatech.com</a>
<u>Bug Finding</u>	
FindBugs	<a href="http://www.findbugs.org">http://www.findbugs.org</a>
Coverity	<a href="http://www.coverity.com">http://www.coverity.com</a>
Visual Studio 2005 \analyze	<a href="http://msdn.microsoft.com/vstudio/">http://msdn.microsoft.com/vstudio/</a>
Klocwork	<a href="http://www.klocwork.com">http://www.klocwork.com</a>
<u>Security Review</u>	
Fortify Software	<a href="http://www.fortify.com">http://www.fortify.com</a>
Ounce Labs	<a href="http://www.ouncelabs.com">http://www.ouncelabs.com</a>

# Analyzing Source vs Compiled



- Static analysis can examine a program
  - As a compiler sees it (Source code) OR
  - As a run-time env sees it (in some cases – bytecode or executable)
- Advantages of compiled code analysis
  - No need to guess how compiler will interpret
  - Source code may be not available
- Disadvantages
  - Making sense is more difficult (e.g., may lack type info)

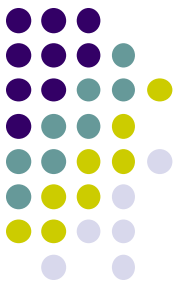
# SA in Code Review





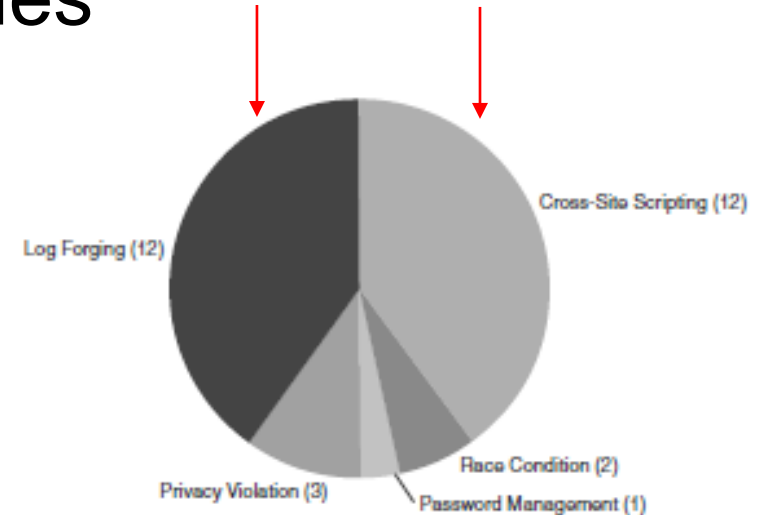
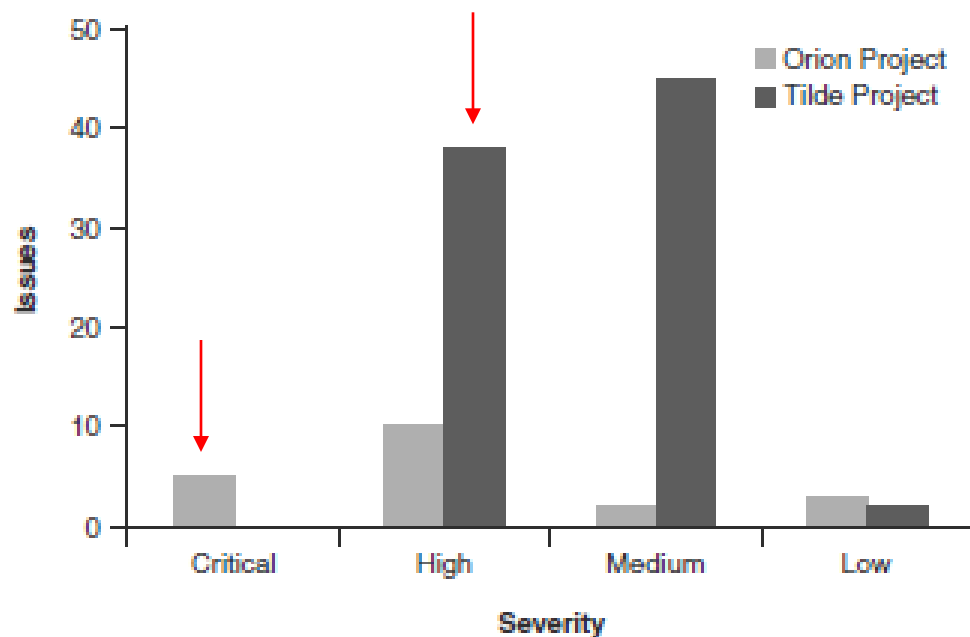
# Establish Goals: SA Metrics

- Prioritize code to review + criteria ... based on risks
- Metrics helps
  - Prioritizing remedial efforts
  - Estimating risk associated with code (tricky!)
    - False positive/negative – manual inspection needed
    - No way to sum/aggregate risks from flaws
- Some metrics for tactical focus
  - Measuring vulnerability density
    - #results/LOC – maybe deceptive
  - Comparing projects by severity
  - Breaking down results by category
  - Monitoring trends – from one group (dev) to another (security)



# SA Metrics

- Comparing modules based on severity
- Breaking down by categories

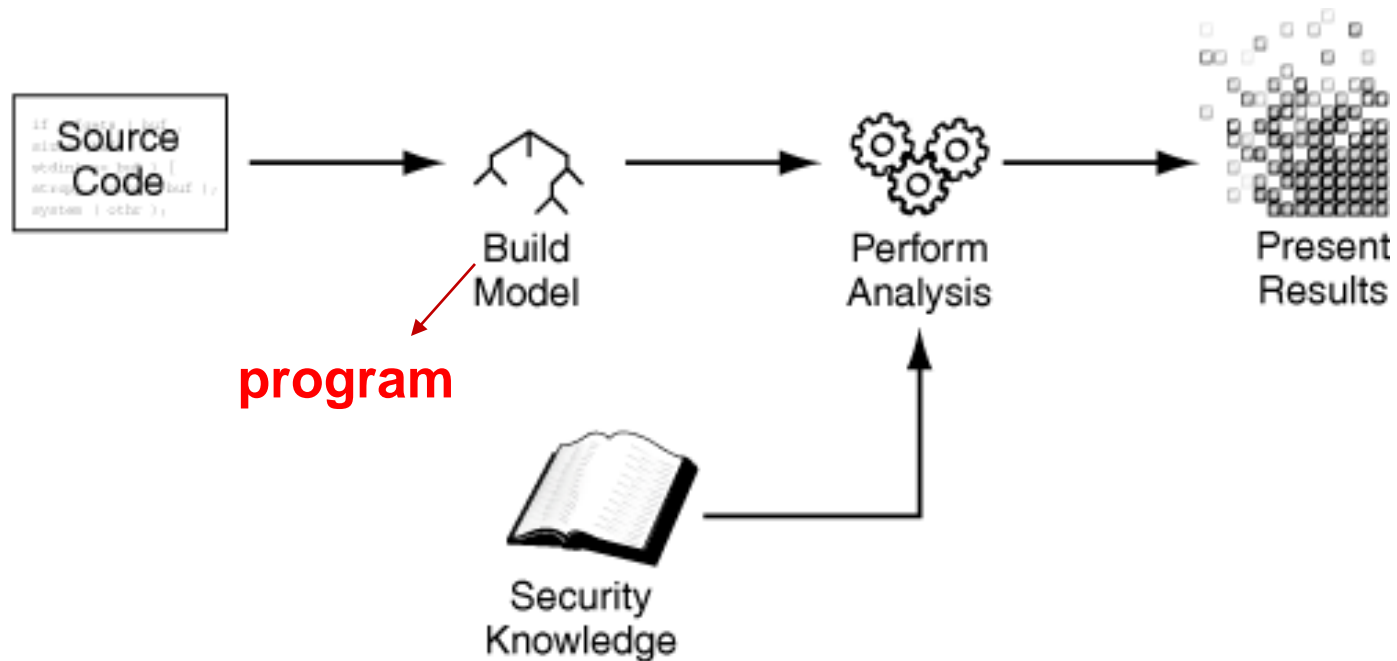


Prioritizing remedial efforts

# SA Internals



- A Generic SA Tool







# Building a model

- Create a **program model** from code
  - A set of data structures
  - Depends on the type of analysis that a tool performs
- SA - Closer to compiler
  - Lexical analysis – e.g., regular expression for tokens
  - Parsing – uses a context free grammar
    - Set of production rules
    - Parse tree: Lex and Yacc

## Lexical Rules:

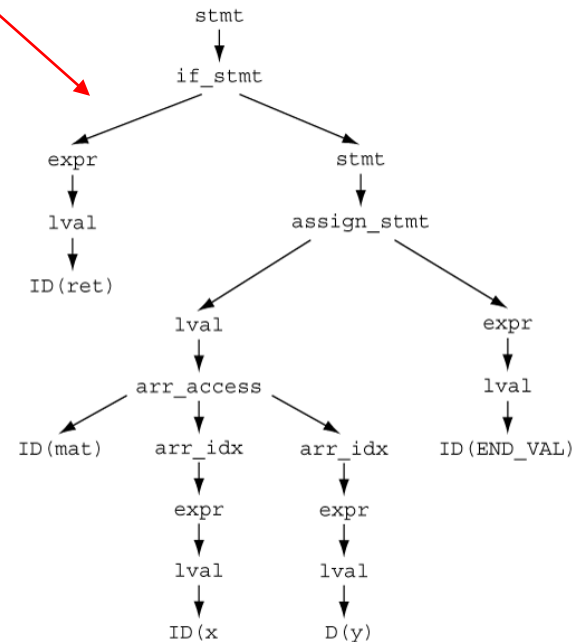
```
if          { return IF; }
(           { return LPAREN; }
)           { return RPAREN; }
[           { return LBRACKET; }
]           { return LBRACKET; }
=           { return EQUAL; }
;           { return SEMI; }
/[ \t\n]+/  { /* ignore whitespace */ }
/^V.*/      { /* ignore comments */ }
/[a-zA-Z][a-zA-Z0-9]*/ { return ID; }
```



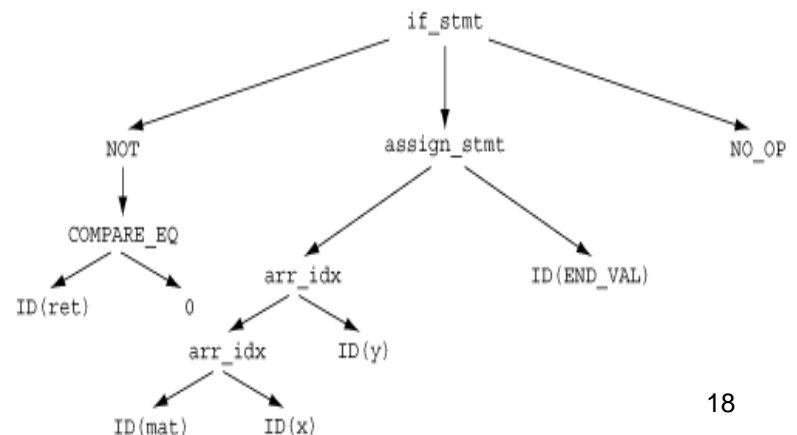
```
if (ret) // probably true
  mat[x][y] = END_VAL;
```

# Parsing

- Can have nonterminal symbols
  - Syntactic sugar!
- Can perform analysis on Parse Tree – can be inconvenient
  - Directly from grammar



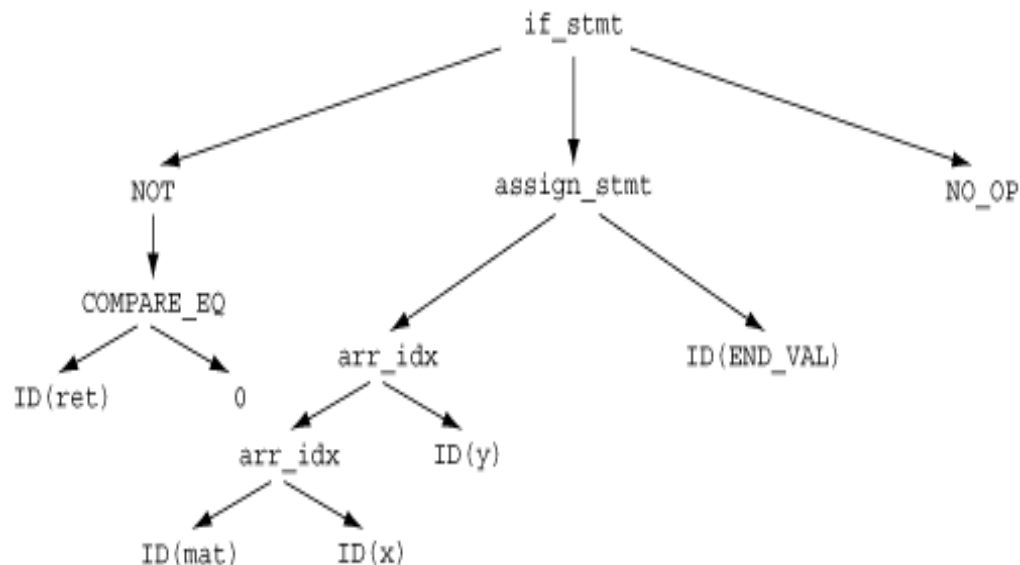
```
stmt := if_stmt | assign_stmt
if_stmt := IF LPAREN expr RPAREN stmt
expr := lval
assign_stmt := lval EQUAL expr SEMI
lval = ID | arr_access
arr_access := ID arr_index+
arr_idx := LBRACKET expr RBRACKET
```





# Abstract Syntax Tree

- Does away with the details of grammar and *syntactic sugar*
  - Create a standard version of program
  - *Lowering* (e.g., loops may be converted to while loop)



# Semantic Analysis & Control Flow



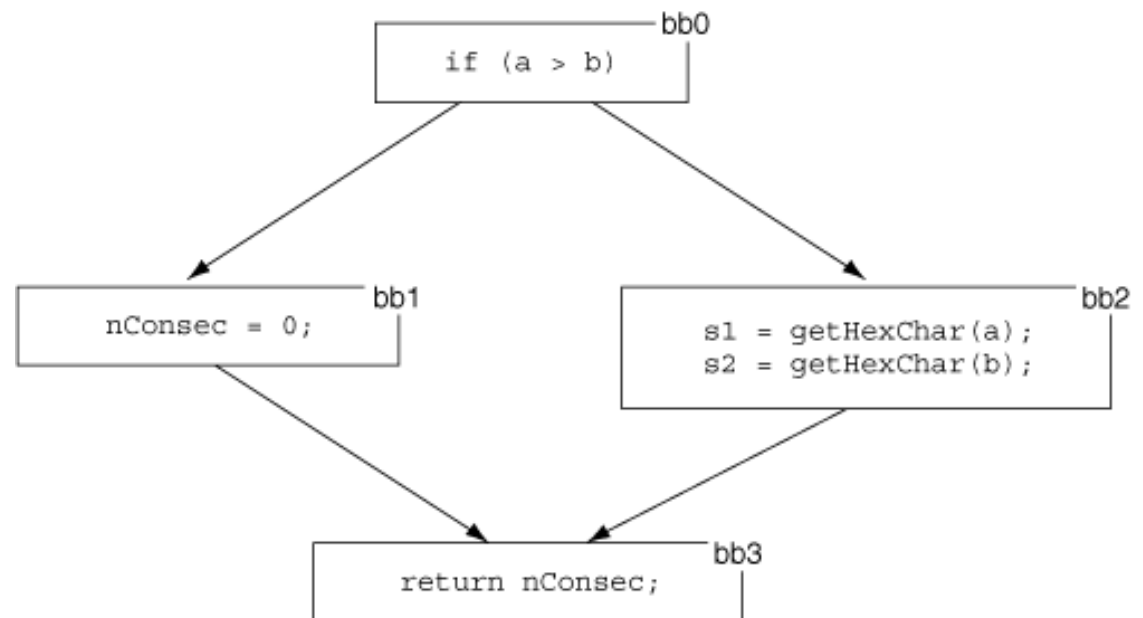
- Semantic analysis based on: AST + Symbol table
  - Type checking can be done
  - Semantic analysis – symbol resolution and type checking
  - Optimization or *intermediate* forms may be created
- Tracking Control Flow
  - Different execution paths need to be explored
  - Build a control flow graph on top of AST



# Control Flow Graph

- **Trace**: sequence of **blocks** that define a path
  - E.g., bb0, bb1, bb3

```
if (a > b) {  
    nConsec = 0;  
} else {  
    s1 = getHexChar(1);  
    s2 = getHexChar(2);  
}  
return nConsec
```

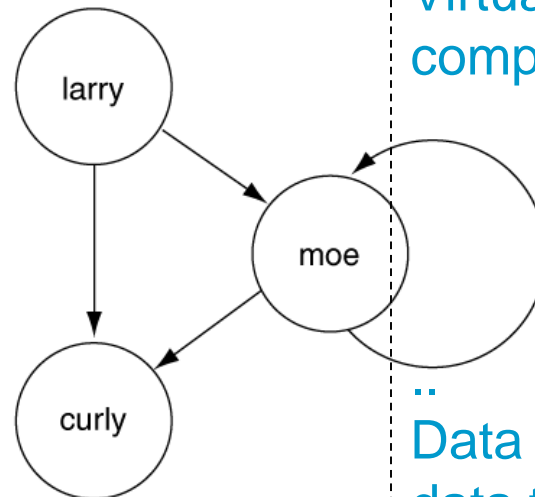




# Call graph

- Call graph – control flow between functions

```
int larry(int fish) {  
    if (fish) {  
        moe(1);  
    } else {  
        curly();  
    }  
}  
  
int moe(int scissors) {  
    if (scissors) {  
        curly();  
        moe(0);  
    } else {  
        curly();  
    }  
}  
  
int curly() {  
    /* empty */  
}
```



Function pointers &  
Virtual functions  
complicate things

..  
Data flow &  
data type  
analysis  
may be needed

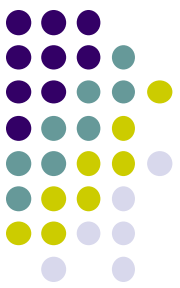
Dynamically  
loaded  
modules  
make it  
further  
challenging

Call graph  
may be  
incomplete



# Dataflow

- Analyzes how data move through the program ..
  - Helps compilers optimize!
- Traverse function's control flow graph
  - Where data values are generated & where used
  - Convert a function to *static single assignment* form (SSA)
    - SSA: allows assigning a value to a variable only once
      - New variables may need to be added
    - SSA variable can have a *constant* (use that to replace future variable places) – *constant propagation* (pwds?, keys)
    - SSA variable may have different values along different control paths – need to be reconciled
      - Merge point:  $\phi$ -function



# SSA Examples

Regular source code form:

```
sum = sum + delta ;
sum = sum & top;
y = y + (z<<4)+k[0] ^ z+sum ^ (z>>5)+k[1];
y = y & top;
z = z + (y<<4)+k[2] ^ y+sum ^ (y>>5)+k[3];
z = z & top;
```

SSA form:

```
sum2 = sum1 + delta1 ;
sum3 = sum2 & top1;
y2 = y1 + (z1<<4)+k[0]1 ^ z1+sum3 ^ (z1>>5)+k[1]1;
y3 = y2 & top1;
z2 = z1 + (y3<<4)+k[2]1 ^ y3+sum3 ^ (y3>>5)+k[3]1;
z3 = z2 & top1;
```

Regular source code form:

```
if (bytesRead < 8) {
    tail = (byte) bytesRead;
}
```

SSA form:

```
if (bytesRead1 < 8) {
    tail2 = (byte) bytesRead1;
}
tail3 =  $\phi$ (tail1, tail2);
```





# Taint Propagation

- It is important
  - to identify which values in a program an attacker could potentially control/target
    - Need to know where values enter and how they move
      - E.g., Buffer overflow vulnerability
  - *Taint propagation* algorithm
    - Key to identifying many input validation and representation defects
    - Static as well as dynamic *taint propagation* analysis



# Pointer Aliasing

- Several pointers may refer to the same memory

\*p1 = 1

Can p1 and p2 refer to the same location?

\*p2 = 2

Can these be reordered?

For the following, compiler should understand that input data flows to process Input

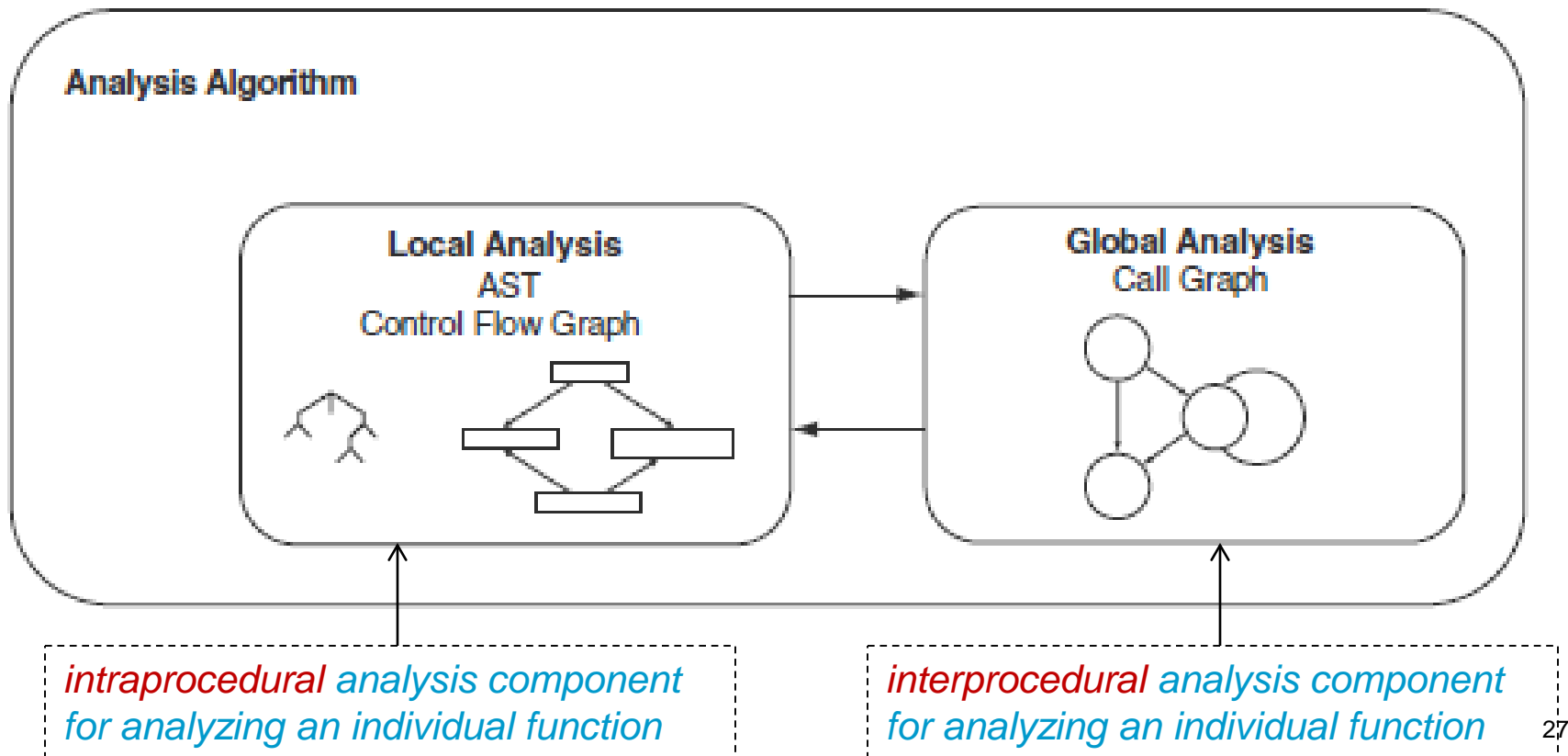
```
p1 = p2;
```

```
*p1 = getUserInput();
```

```
processInput(*p2);
```

# SA Algorithms

- Local component and global component
  - Improve context sensitivity





# Assertions

- Many properties can be specified as assertions
  - which need to be true

Example: Buffer Overflow prevention check

```
strcpy(dest, src);
```

Add assertion before the call

```
assert(alloc_size(dest) > strlen(src));
```

- If there are conditions under which an assertion can fail – report **potential overflow**



# Assertions

- Typically three varieties of assertions
  - Taint propagation problems
    - When programmers trust input when they should not – so SA should check data values moving
    - data is either tainted (controlled by an attacker) or not
  - Range Analysis
    - To Identify buffer overflow – need to know the size of the buffer and the data value
      - Understand the range of values data or size may have
  - Type state: concern about the state of an object as execution proceeds
    - In freed state (can lead to double free vuln?)

# Naïve Local Analysis (informal)



No concrete  
values  
needed

Consider       $x = 1;$   
                   $y = 1;$   
                   $\text{assert}(x < y);$

$x = v;$   
 $y = v;$   
 $\text{assert}(x < y);$   
**Same Result**

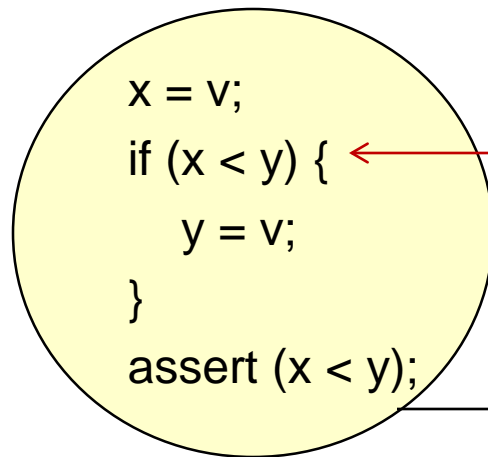
**Symbolic  
Simulation**

- Maintain facts **before** each statement is executed

$x = 1;$	$\{\}$ (no facts)
$y = 1;$	$\{ x = 1 \}$
$\text{assert}(x < y);$	$\{ x = 1, y = 1 \}$

- **Always false!!** SA should report a problem

# Conditionals make it complex!



this condition may or may not be TRUE

x = v;	{ } (no facts)
if (x < y)	{ x = v }
assert (x < y)	{ x = v, $\neg(x < y)$ }

When BRANCH is not taken  $x < y$  is FALSE

x = v;	{ } (no facts)
if (x < y)	{ x = v }
y = v;	{ x = v, $x < y$ }
assert (x < y)	{ x = v, $x < y$ , $y = v$ }

When BRANCH is taken  $x < y$  is TRUE

$v < v$  means assertion  
is violated

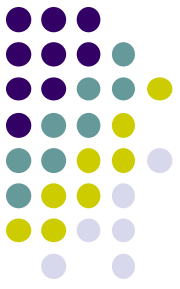
Need to check the  
conjunction of assertion  
predicate and all the facts:

$(x < y) \wedge (x = v) \wedge \neg(x < y)$

Again fails!

# Conditionals make it complex!

## Loops add further ..



- The previous approach is problematic
- #paths grows with the number of conditionals
  - Share info among common subpaths
  - *Program slicing* – to remove code that cannot affect the outcome of the assert predicate
  - Also eliminate *false paths* – logically inconsistent paths that will never be executed
- Adding loops makes it even more complex!



# Approaches to Local Analysis



- Abstract interpretation
  - Abstract away aspects of the program that are not relevant to properties of interest and then **perform an interpretation**
  - Loop problems – do **flow-insensitive** analysis
    - Tries to guarantee that all statement orderings are considered (not follow the program statement order)
      - No need for control flow analysis
      - But some useless execution order may be performed as well
    - More practical tools – partially flow sensitive!



# Predicate Transformers

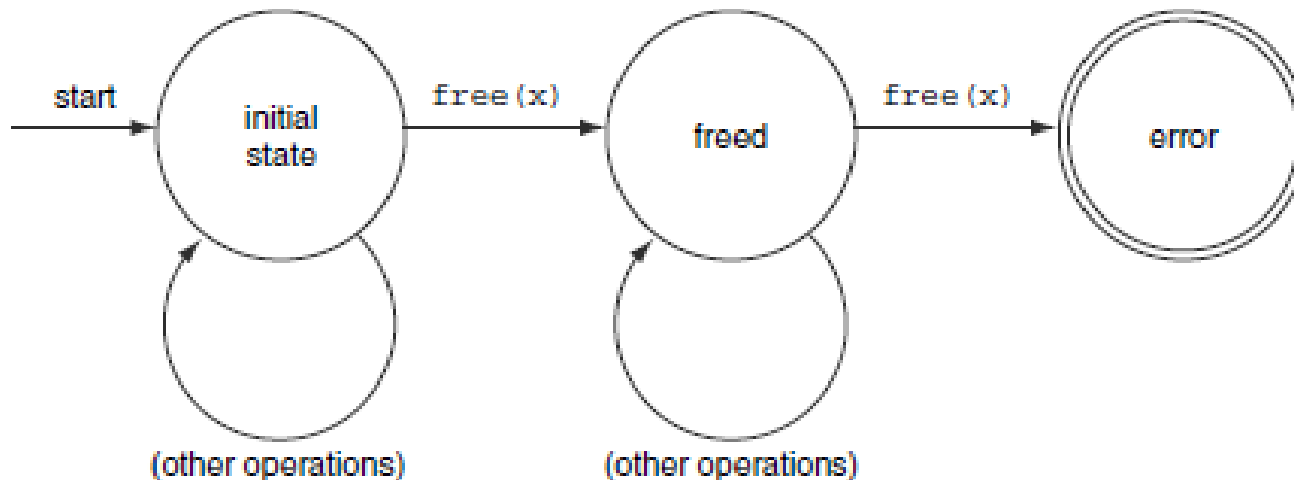
- Use the weakest precondition
    - Fewest set of requirements on the callers of a program that are necessary to arrive at a desired final state or post condition
- E.g., consider `assert(x < y)`

$(x < 0 \wedge y > 0)$  // always satisfied  
is a strong requirement than  
 $(x < y);$



# Model Checking Approach

- Accepts properties as specifications, transforms the program to be checked into an automaton (called the model)
  - Now compare the specification to the model
  - Example: “memory should be freed only once”



**Model checking will look for a variable wrt which system will reach state error**



# Global Analysis

- Context-sensitive analysis
  - Takes into account the context of the calling function
- Whole-program analysis
  - Tries to analyze every function with a complete understanding of the context of its calling functions
  - One way is “*inlining*” (Recursion will be problem)
  - Time consuming and very ambitious
- More flexible approach
  - Local analysis generates the *function summaries*

● Example →

---

```
memcpy(dest, src, len) [  
  requires:  
    ( alloc_size(dest) >= len ) ∧ ( alloc_size(src) >= len )  
  ensures:  
    ∀ i ∈ 0 .. len-1: dest[i] == src[i]  
]
```

---

# Rules

- Good SA tools externalize the rules they check
  - Added, removed, altered easily

RATS will report a violation of the rule whenever it sees a call to `system()` where the first argument is not constant.

**The argument number**

```
<Vulnerability>
  <Name>system</Name>
  <InputProblem>
    <Arg>1</Arg>
    <Severity>High</Severity>
  </InputProblem>
</Vulnerability>
```

In some cases rules are *annotated* within the program (in JML)

```
/*@ public normal_behavior
   @   requires    valid;
   @   assignable  state;
   @   ensures     -1 <= \result && \result <= 65535;
   @*/
public int read();
```



# Rules for Taint Propagation

- Variety of rule types to accommodate different *taint propagation* problems
  - **Source rules** define program locations where tainted data enter the system.
    - Functions named `read()` often introduce taint in an obvious manner; others: `getenv()`, `getpass()`, `gets()`.
  - **Sink rules** define program locations that should not receive tainted data.
    - For SQL injection in Java, `Statement.executeQuery()` is a sink.
    - For buffer overflow in C, assigning to an array is a sink, as is the function `strcpy()`



# Rules for Taint Propagation

- **Pass-through** rules define the way a function manipulates tainted data.
  - E.g.,, a pass-through rule for the `java.lang.String` method `trim()` might explain “if a `String s` is tainted, the return value from calling `s.trim()` is similarly tainted.”
- **Cleanse rule** is a form of pass-through rule that removes taint from a variable.
  - represents input validation functions.
- **Entry-point rules** (similar to source)-
  - they introduce taint into the program, entry-point functions are invoked by an attacker.
    - E.g., `main()` is an entry point (java, C)

# Example: Command injection vulnerability



```

  ①
if ( fgets ( buf , sizeof(buf), stdin) == buf ) {
    ②
    ③
    strcpy ( othr , buf );
    ④
    ⑤
    system ( othr );
}

```

- ① A source rule for `fgets()` taints `buf` `othr`
- ② Dataflow analysis connects uses of `buf`
- ③ A pass-through rule for `strcpy` taints
- ④ Dataflow analysis connects uses of `othr`
- ⑤ Because `othr` is tainted, a sink rule for `system()` reports a command injection vulnerability





# Taints

- Essentially BINARY attribute
  - But can have taint flags to indicate variety of tainted data – can help prioritize!
    - FROM\_NETWORK data from network
    - FROM\_CONFIGURATION data from config file
    - Sink functions may be dangerous for a specific taint type
      - E.g., arbitrary user-controlled data vs. numeric data
- Taint propagation rules include various elements
  - Method or function – to apply to
  - Precondition – on taint propagation
  - Postcondition – changes to taint propagation (taint or cleanse)
  - Severity – when the sink rule is triggered

# Summary

- Overview of Static Analysis

