Code Scanning

David Basin

ETH Zurich
Security Engineering

Some questions

1. You are responsible for quality assurance for a large-scale IT system (10^6 loc)
   - What do you have your team do?
   - Follow coding standards? Code review? Formal verification?

2. Your system is safety or security critical
   - What changes from #1?
   - And does distinction **safety** versus **security** matter?

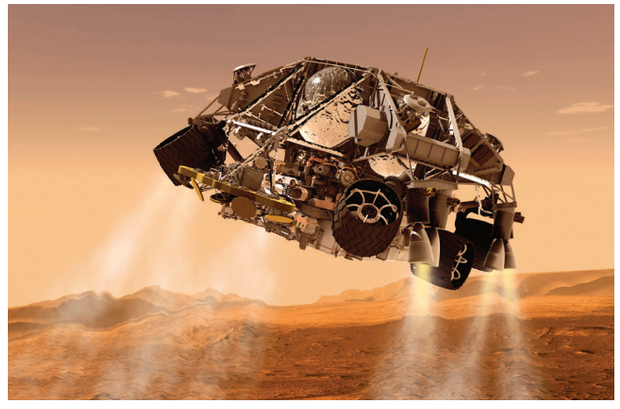
3. You are a researcher building code analysis tools.

   How do you migrate them to large-scale applications?

Problem context

What are best practice?

Critical system example



- Mars curiosity rover: 500K+ lines of code
- **Critical**: small mistakes means loss of mission
  - Moreover, missions are increasingly complex
- NASA takes this extremely seriously
  - Hired many top researchers
  - Extensive Quality Assurance program
- Central part of Quality Assurance: **Risk-Based Coding Standards**
  - 6 Levels of Compliance (LOC)
  - More critical ⇒ higher level

Source “Mars Code”, CACM Feb. 2014

NASA Levels of Compliance

- **LOC 1:** Language compliance, C-language standard ISO-C99
  - So code cannot rely on any compiler extensions
- **LOC 2:** Embedded system requirements
  - **Example**: all loops have statically verifiable upper bounds on number of iterations possible
- **LOC 3:** Density of assertions in code is > 2%
  - Assertions used for testing
  - They remain enabled during missions
  - Failing assertions tied with fault-protection system whereby spacecraft goes into a predefined safe state

NASA Levels of Compliance (cont.)

- **LOC 4:** target level for mission-critical code (all on-board software)
  - Restrictions to CPP, function pointers, and pointer indirection
  - 31 coding rules

- **LOC 5 & 6**: Safety-critical requirements
  - All rules from MISRA C coding guidelines (not covered previously)
    MISRA = Motor Industry Software Reliability Association
  - 143 rules (checkable using static program analysis)
    + 16 "directives" concerning procedural manners.
  - **Example**: do not use malloc, as it may fail
  - All MISRA rules checkable by Coverty and Code Sonar

Code scanning at NASA

- Multiple code scanners used: Coverity, Codesonar, Semmle
  - ✦ All reported flaws are logged
- Supplement manual code reviews. Flaws also logged.
- Developers must respond to, or remediate, flaws found by both!
- Some statistics
  - ✦ 2008-2012: 10,000 peer comments, 30,000 scanner comments
  - ✦ Ca. 84% led to code changes
  - ✦ <2% difference in rate between peer generated and tool generated
  - ✦ Ca. 12% end up in explicit disagree responses

NASA conclusions

- Extremely positive experience with scanning
  "**A static analyser will not tire of checking for the same types of defects over and over, night after night, patiently reporting all violations**"
- They also use model-checkers (Spin)
  - Used for selected critical components, e.g. with tricky concurrency
  - Automatically extract models from C source-code in some cases
  - Manually constructed models in others
- Excellent results: system reliability (now) very high.

Code analysis and security

A trivial example

- C function to print a message to a given file descriptor

```
void printMsg(FILE* file, char* msg) {
  fprintf(file, msg);
}
```

- Defensive version

```
void printMsg(FILE* file, char* msg) {
  if (file == NULL) {
    logError("attempt to print message to null file");
  } else if (msg == NULL) {
    logError("attempt to print null message");
  } else {
    fprintf(file, msg);
  }
}
```

- Is this now OK?

What can an adversary do?

- Could enter: AAA1_%08x.%08x.%08x.%08x.%08x.%n
  - ✦ This is a “format string” exploit, analogous to a buffer overflow.
  - ✦ Input seems absurd. But this doesn’t stop the adversary
  - ✦ Inputs from untrusted sources can always be **tainted**
- Solution here

```
void printMsg(FILE* file, char* msg) {
  if (file == NULL) {
    logError("attempt to print message to null file");
  } else if (msg == NULL) {
    logError("attempt to print null message");
  } else {
    fprintf(file, "%.128s", msg);
  }
}
```

- Programmers easily make such errors

  Code scanners should help find them!

Scope: static analysis applied to implementation code not design

	Visible in the code	Visible only in the design
Generic defects	Static analysis sweet spot. Built-in rules make it easy for tools to find these without programmer guidance. • *Example: buffer overflow.*	Most likely to be found through architectural analysis. • *Example: the program executes code downloaded as an email attachment.*
Context-specific defects	Possible to find with static analysis, but customization may be required. • *Example: mishandling of credit card information.*	Requires both understanding of general security principles along with domain-specific expertise. • *Example: cryptographic keys kept in use for an unsafe duration.*

Problem categories (from security perspective)

1. **Input validation and representation**

    - Buffer overflows, cx-scripting, injection attacks, etc.

2. **API abuse**: abuse contract between caller and callee

    - Provide wrong input or make too strong assumptions about output

3. **Security features**: e.g., don't hardcode passwords in source code

4. **Time and state**: e.g., race condition example

5. **Error handling**: handling errors poorly or not at all

6. **Code quality**: dereference null pointers, infinite loops, etc.

7. **Encapsulation**: lack of it

What is code scanning?

Scope

- **Rough definition**: **code scanning = pragmatic static analysis**

  Analyze source code. No testing, runtime verification, etc.

- As we will see, p**ragmatism is essential**

- Based on research community successes
  - Type checking
  - Property checking (model-checking)
  - Abstract interpretation
- Also on software engineering best practices
  - Style checking, program comprehension, security reviews, etc.

Let's consider a few examples

Type checking

- The Java compiler will flag this as an error. Is it?

```
short s = 0;
int i = s;
short r = i;
```

- How about this?

  What will happen at runtime?

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

- Type checkers useful
  - ✦ But may suffer from false positives/negatives
  - ✦ Identifying which computations are harmful is undecidable

Style checking

- Enforce more picker and more superficial rules than type checkers
- Some compilers can check these

  E.g., gcc -Wall checks cases

```
 1 typedef enum { red, green, blue } Color;
 2
 3 char* getColorString(Color c) {
 4  char* ret = NULL;
 5  switch (c) {
 6    case red:
 7      printf("red");
 8  }
 9  return ret;
10 }
```

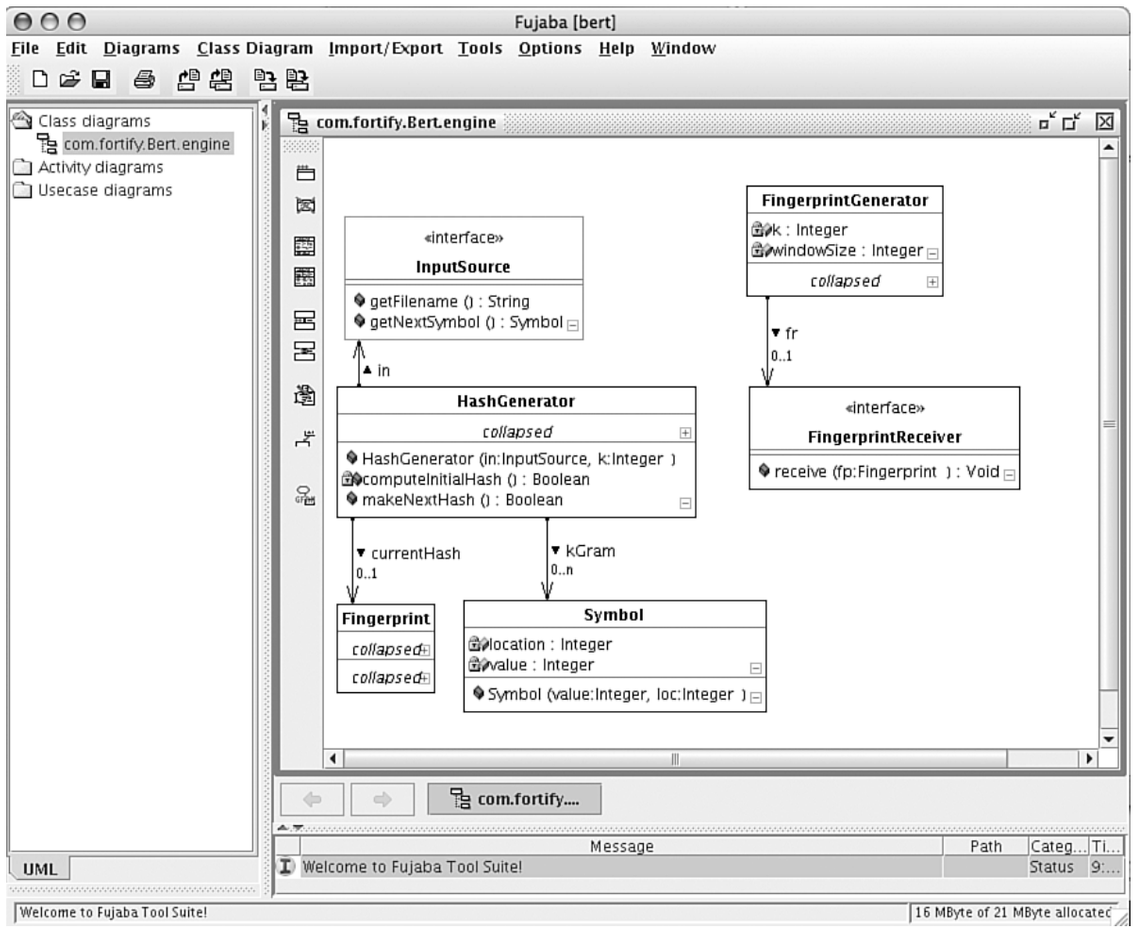
```
enum.c:5: warning: enumeration value 'green' not handled in switch
enum.c:5: warning: enumeration value 'blue' not handled in switch
```

- Style checkers often extensible

  PMD for Java is a good example of this
- Don’t laugh! Consider NASA experience

Program understanding

- Tools can help with
  - ✦ Making sense of large code base
  - ✦ Reverse engineering abstractions
  - ✦ Code slicing
  - ✦ Finding declarations and uses.



- All this is useful for manual code reviews

Property checking (model-checkers, etc.)

- Check all behaviors against property specifications.
- Consider following code:
  - What property might you require?
  - How could you check this?

```
inBuf = (char*) malloc(bufSz);
if (inBuf == NULL)
  return -1;
outBuf = (char*) malloc(bufSz);
if (outBuf == NULL)
  return -1;
```

- Model-checkers are powerful. But:
  - State space explosion is a problem
  - You must specify properties and often the model too
- Some industrial use, e.g., NASA, Intel, …

Bug (pattern) finders

- Work with a fault model of typical programmer mistakes

```
1  Person person = aMap.get("bob");
2  if (person != null) {
3      person.updateAccessTime();
4  }
5  String name = person.getName();
```

Null-pointer dereference

```
1  String b = "bob";
2  b.replace('b', 'p');
3  if(b.equals("pop"))
```

Ignored return value

- Note overlap with other methods
- Findbugs is best example of such a tool

Why is it hard?

Theory and practice

Why is it hard in theory?

- **Flaws** concern problematic behavior
  - **Failures**: deviation of behavior detectable at system interface
  - **Errors**: deviation of system's behavior from intended one
- Behavioral properties in general undecidable.
  - Termination, reachability of a program point, …
- Tools must therefore
  - **not always terminate**, or
  - **over-approximate** behavior (returning **false positives**), or
  - **under-approximate** behaviors (returning **false negatives**)

And in practice?

Coverty experience ("A few billion lines of code later", CACM, 2010)

- Tool finds generic errors and interface-specific violations

  E.g., memory corruption, data races, function-ordering constraints, …

- Company's goals: build user community, $$$

- **Central religion is results**: good = works!
  - **Find errors**, not prove their absence!
  - **Embrace unsoundness!** Focus on low-hanging fruit!

    "Circa 2000, unsoundness was controversial in the research community, though it has since become almost a de facto tool bias for commercial products and many research projects"
  - **Usability** and **simplicity** are critical! For both sales and deployment

Coverty experience

“Theory and practice are similar, but only in theory”

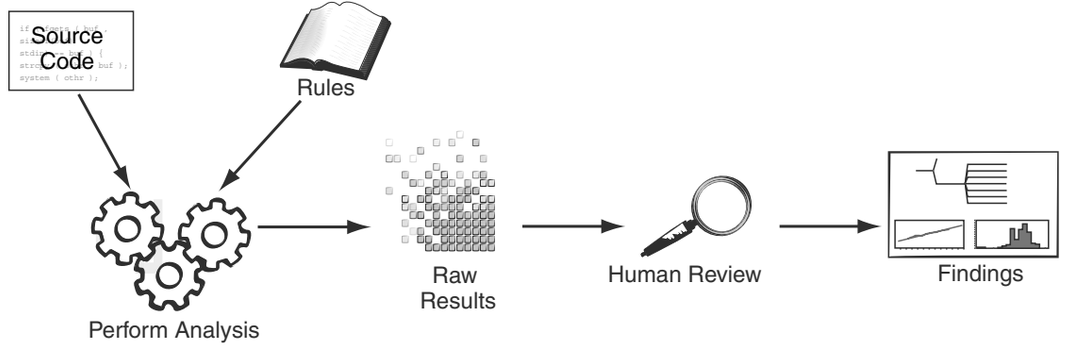
- Hardest part is to find the code
  - ✦ Home-brown build environments, special tools make problem hard
  - ✦ Solution: run build process and intercept all system calls
  - ✦ But system must now build!
- And you have to parse it
  - ✦ Experience: the C language doesn’t exist. Same for C++, Java, …
- Customers must understand the bugs and care about them

  Give informative output, rank severity and confidence, etc.
- No churn: tool output shouldn’t change much, even after upgrades!
- Avoid false positives. More than 30% ⇒ tool won’t be used/sell.

Basic dilemma: cry wolf vs. missing detail kills!

How are code scanners used?

Code review cycle



- Scanners used within a code review

  Set goals, run tools, review code, fix it

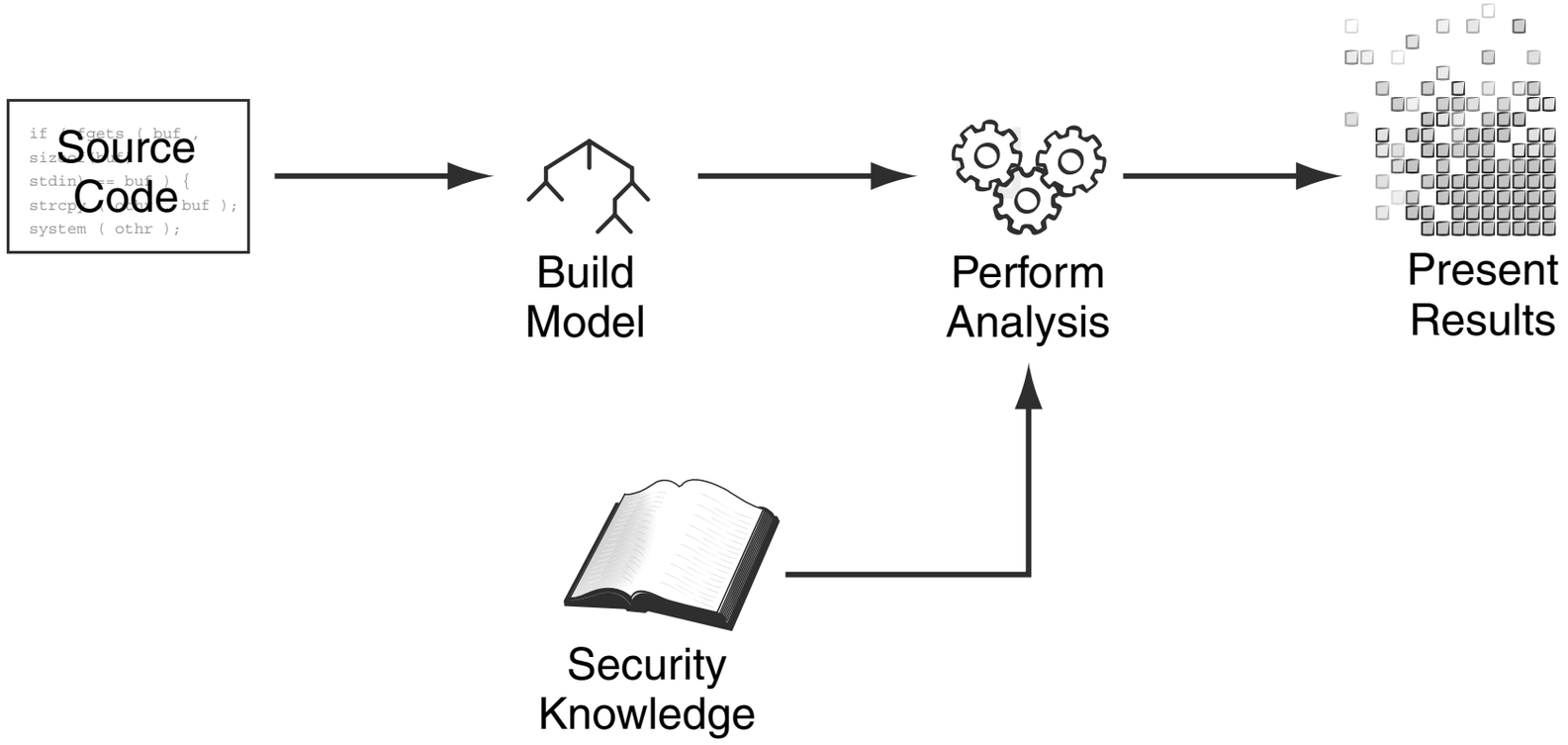
- Often augmented by adding new rules to scanner
  - ✦ Rules encoding coding-style/company/product specific rules
  - ✦ Rules motivated by problems not found by scanner
- For fixes, developers must first be convinced. **Exploitability trap.**
  - ✦ Often result is not unambiguously secure or exploitable
  - ✦ Difficult to motivate improvements. Especially for style issues.
- **Code review should be more than scanning!** **Why?**

In projects, embed code scanning within code review!

Code scanning basics

Introduction to (parts of) the tool bag

Overview



- We use Fortify tool as rough “tool model”

  See Chess & West “Secure Programming with Static Analysis”

- Provide abstract account here

- Afterwards, examine specialized techniques for different bugs

- Focus on bugs that represent **security vulnerabilities**

Model-building — from source to AST

Standard from compiler design

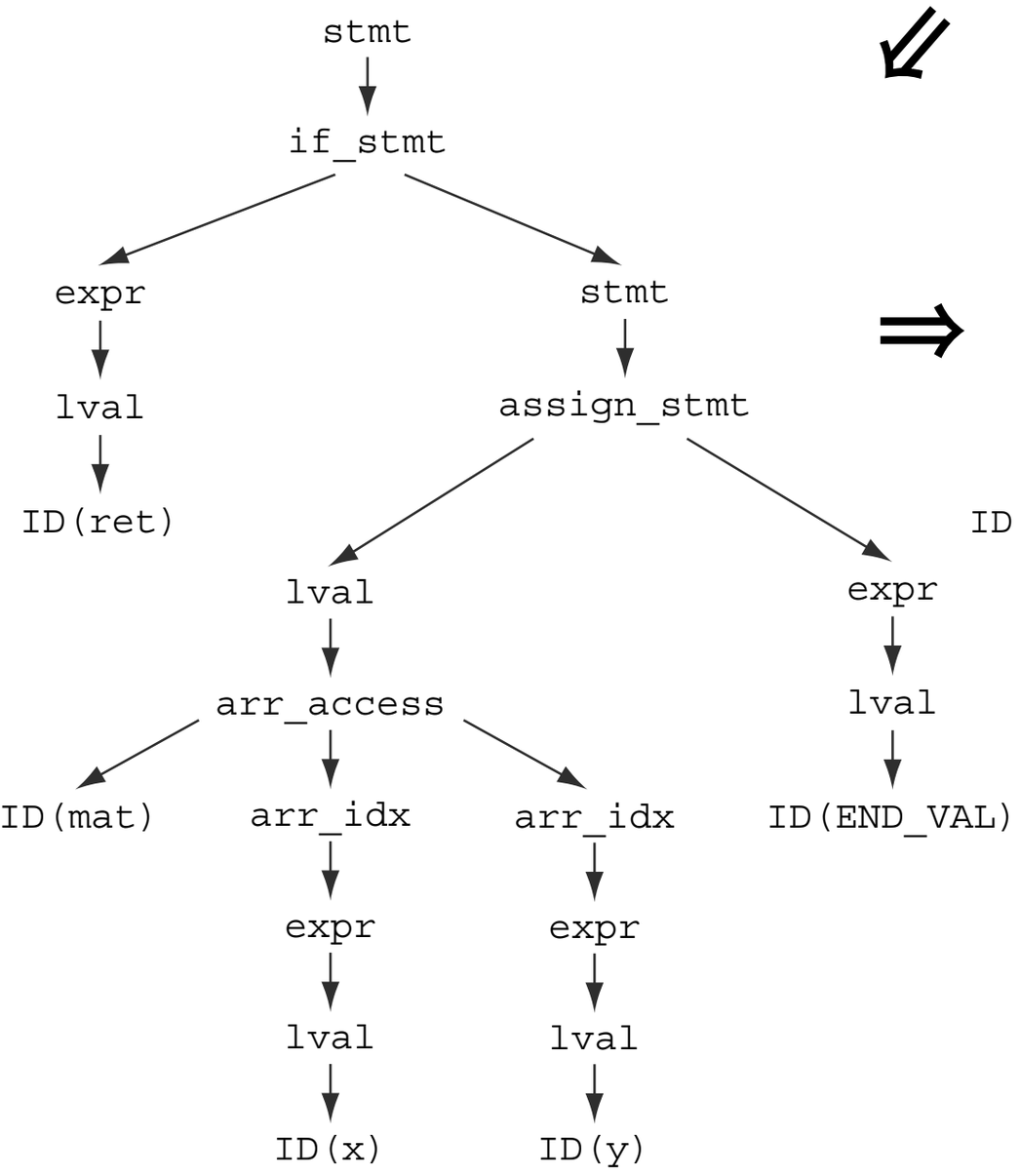
```
if (ret) // probably true
  mat[x][y] = END_VAL;
```

⇒

```
IF LPAREN ID(ret) RPAREN ID(mat) LBRACKET ID(x) RBRACKET LBRACKET
ID(y) RBRACKET EQUAL ID(END_VAL) SEMI
```

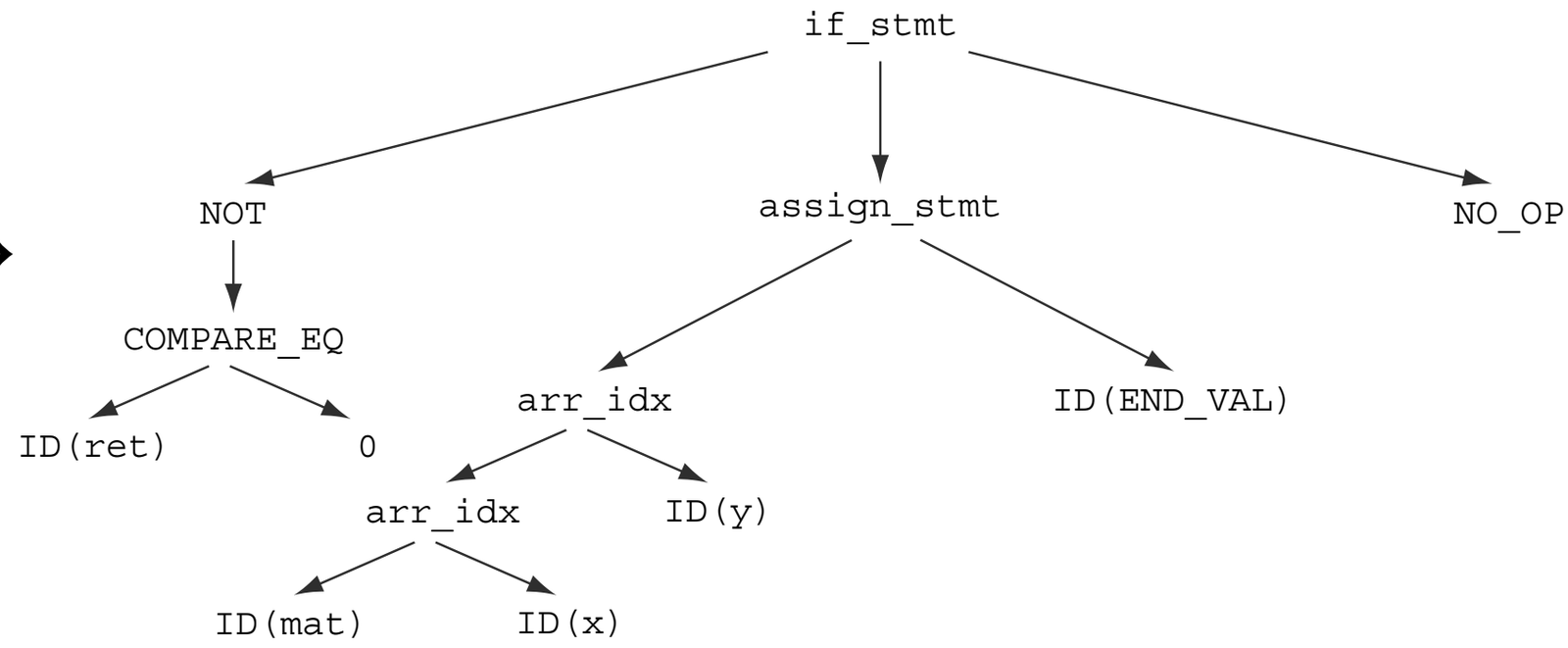
Source

Token stream



Parse tree

⇒



Abstract syntax tree

Abstracts away details of grammar and makes language aspects explicit (e.g., true is .neq 0).

Can also simplify language, e.g., only one kind of loop.

Semantic analysis and structural rules (I)

- Build symbol table along with AST (as in compilers)
- This can be used for type checking, which helps find bugs
- Types also help in specifying “structural rules” for bug finding
- **Example**: Java DB connections should not be shared between threads
  - So DB connections should not be stored in static fields
  - To find instances of this problem look for

    Field: static and type.name = “java.sql.Connection”
  - Above is an example of a Fortify “structural rule”

Semantic analysis and structural rules (II)

- **Example: buf = realloc(buf, 256)**
  - Memory leaks If **realloc()** fails and returns **null**
  - Can detect with following structural rule

    FunctionCall c1:

    c1.function is [name == “realloc”] and

    c1 in [AssignmentStatement: rhs is c1 and

    lhs == c1.arguments[0]

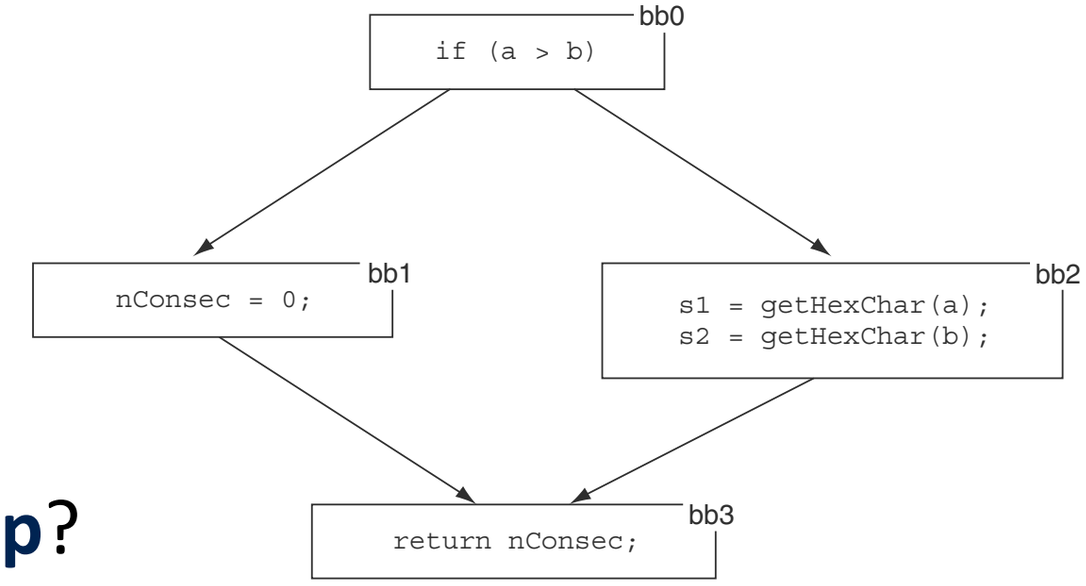
- So some overlap between compilers and code scanners
  - Compiler type checking **is** code scanning
  - Example **lint:** finds suspicious issues and portability bugs for C

Going beyond…

- Previous examples were simple
  - Compare type with field kind
  - Compare lhs with argument of function call
- To go beyond this need to reason about
  - **control flow**: how instructions are sequenced
    - e.g., are acquired locks released?
  - **data flow**: how data moves through program variables
    - e.g., does input come from a network-facing interface?

Let’s look at **basic approaches** to these problems

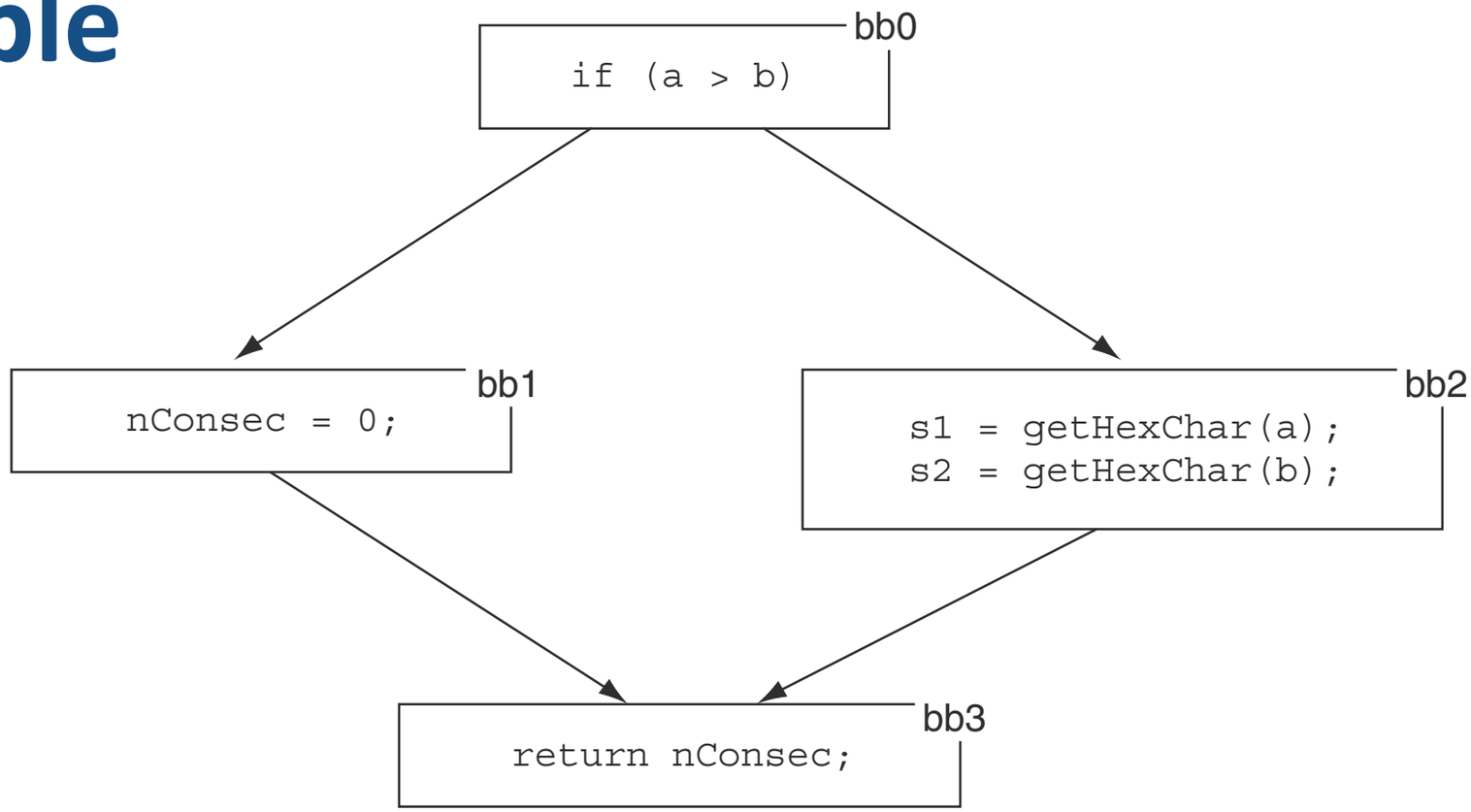
Control flow



- Need to answer questions like
  - Which execution paths lead to point **p**?
  - Which execution paths follow from **p**?
- Tools build **control flow graph** on top of AST
  - **Basic block**: sequence of instructions that is always executed, i.e., no jumps in/out of middle, no branching, etc.
  - **Forward edges**: potential control flow paths between BBs
  - **Backward edges**: possible loops
- System **traces i**n terms of BB sequences

Control flow example

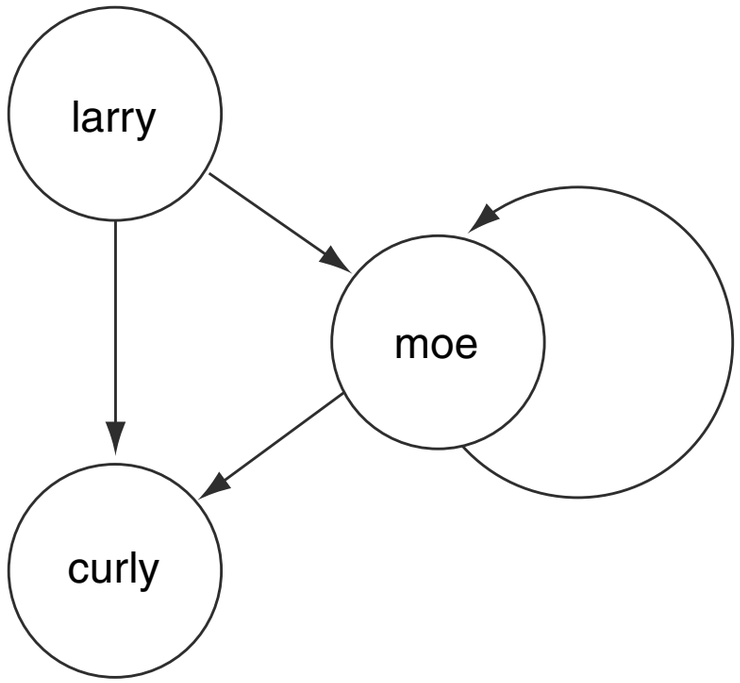
```
if (a > b) {
  nConsec = 0;
} else {
  s1 = getHexChar(1);
  s2 = getHexChar(2);
}
return nConsec;
```



- In practice, graph nodes store pointers to AST nodes
- Two traces: [bb0, bb1, bb3] and [bb0, bb2, bb3]
- **Gretschenfrage** for bug finders:

  report bugs in **some** versus **all** traces?

Function call graph




```
int larry(int fish) {
  if (fish) {
    moe(1);
  } else  {
    curly();
  }

}

int moe(int scissors) {
  if (scissors) {
    curly();
    moe(0);
  } else {
    curly();
  }
}
```

```
int curly() {
  /* empty */
}
```

- Represents control flow between functions and methods
- Different representations may need to be combined
  - Traces will depend on function calls
  - Even trickier in practice, e.g., with function pointers or virtual methods must also use dataflow analysis and datatype analysis to limit set of potential functions that can be invoked from a call site.

Dataflow analysis

- Determine how data moves through program
  - ✦ Traverse control flow graph and note where data generated and used
- Implementation trick: convert function to Static Single Assignment
  - ✤ Can assign to a variable only once, so make unique with indicies
  - ✤ Makes it trivial to determine where value comes from
- Simple compiler application: constant propagation
  - ✤ If SSA variable assigned a constant, then replace it by constant
- **Question**: any security relevant applications of constant propagation?

  Find hard-coded passwords or encryption keys!

Dataflow examples

```
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;
```

```
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;
```

Example 1 (tiny encryption): simple, since straight-line code

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

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

Example 2 (branching): requires reconciling variable where control paths merge. Done by introducing phoney "selection function" ϕ.

Dataflow application #1: taint analysis

- What values can an attacker potentially control?
- Solve by tracing how his input moves through program
  - Existence of path from input function to a vulnerable operation
  - Vulnerable operation might, for example, be a buffer overflow
- Taint analysis/tracking can also be done dynamically (e.g., Perl)

```
#!/usr/bin/perl -T;  use strict; use warnings;   # Check declarations, typos, etc.

my $arg = $ARGV[0];                                  # 1st command line argument

# Untaint data and check result
$arg =~ m/^([a-zA-Z0-9\._]+)$/ or die "Bad data in first argument";      #match pattern
my $file = "/home/foo/$1";          # pattern match stored in $1, i.e., assign untainted data
open FOO, ">$file" or die $!;
print FOO “output for file\n";
close FOO;
```

If (tainted) input doesn’t match regexp (sanitization), program dies

Dataflow application #2: pointer aliasing

- Determine which pointers point to same memory locations

  Standard for compilers, e.g., can two statements be reordered?

  ```
  *p1 = 1;  *p2 = 2
  ```

- Security application: taint analysis!

  ```
  p1 = p2;
  *p1 = getUserInput();
  processInput(*p2)
  ```

- In practice, analysis algorithms are conservative

  Determine: **must** versus **may** versus **cannot**

Analysis algorithms

Improving power and precision

Need for sophistication

- **Example**: strcpy(dest, src)
  - You should probably never use it. But not all occurrences harmful
  - If **assertion** alloc_size(dest) > strlen(src) always succeeds, no overflow
- Analogous situation for: SQL injections, cross-site scripting, …
- So let's treat code scanning as a property-checking problem!
  - **Question 1:** how do you check properties?
  - **Question 2**: which properties do you check?
    - **Taint propagation**: is value tainted? (How do you formalise this?)
    - **Range analysis**: relevant, e.g., for buffer overflows
    - **Type state**: is variable in right "type" of state at given point?
      E.g., free(p) called in state where p not already freed

Difficulties in assertion checking (simplest case)

- Consider a simple program and its possible analysis

x = 1;	{} (no facts)
y = 1;	{x = 1}
assert (x < y)	{x = y, y = 1} ✘

  - ✦ Conclude 1 < 1 , i.e., report that assertion always fails!
  - ✦ Same conclusion holds if “1” is replaced with “v”

    This is **symbolic simulation** and conclusion holds for all v

- So straightline code is easy! Where do the problems lie?

Assertion checking — branches

x = v;	{}	{}
if (x < y) {	{x = v}	{x = v}
y = v; }	{x= v, x < y}	
assert (x < y)	{x = v, x < y, y = v}	{x = v, ¬(x < y)}
Program	**x < y case** ✗	**¬(x < y) case** ✗

- Evaluate **assertion** with all information
  - Case 1: x = v ∧ **x < y** ∧ y = v. Implies v < v, a contradiction
  - Case 2: **x < y** ∧ x = v ∧ ¬(x < y). Inequalities contradict

    So assertion always fails
- Problem: #paths is exponential in #conditions
  - Problem slicing is one technique to cope with this (not covered here)
  - Loops even more problematic. Solutions?

Assertion checking in practice

- Static assertion checking is verification!
  - ✦ Long history of work on this, both interactive and automated
- Predicate transformers + theorem proving
  - ✦ For program S & predicate R, wp(S,R) is weakest precondition on initial state such that S terminates in a final state satisfying R
  - ✦ R may be desired assertion
  - ✦ Theorem proving is needed in general

$$
\begin{array}{rcl}
wp(\textbf{if } x < y \textbf{ then } x := y \textbf{ else } \textbf{ skip } \textbf{ end},\ x \geq y) & = & (x < y \Rightarrow wp(x := y, x \geq y)) \ \wedge \ (\neg(x < y) \Rightarrow wp(\textbf{skip}, x \geq y)) \\
& = & (x < y \Rightarrow y \geq y) \ \wedge \ (\neg(x < y) \Rightarrow x \geq y) \\
& \Leftrightarrow & \textbf{true}
\end{array}
$$

  - ✦ Loops require inventing loop invariant (or approx. by finite unrolling)
- Model checking, but then state space should be finite

Which properties do you check?

- Research suggest that the **secret sauce** is the properties!
- Different options

1. Let programmer annotate program with assertions or provide rules
2. Use standard collection based on **fault models** or **bug categories**
3. Infer rules based on “consistency”
   If a method is called 100 times and return value is used 99 times, then flag non-checking instance as suspicious

Option 2 appears most popular in practice.

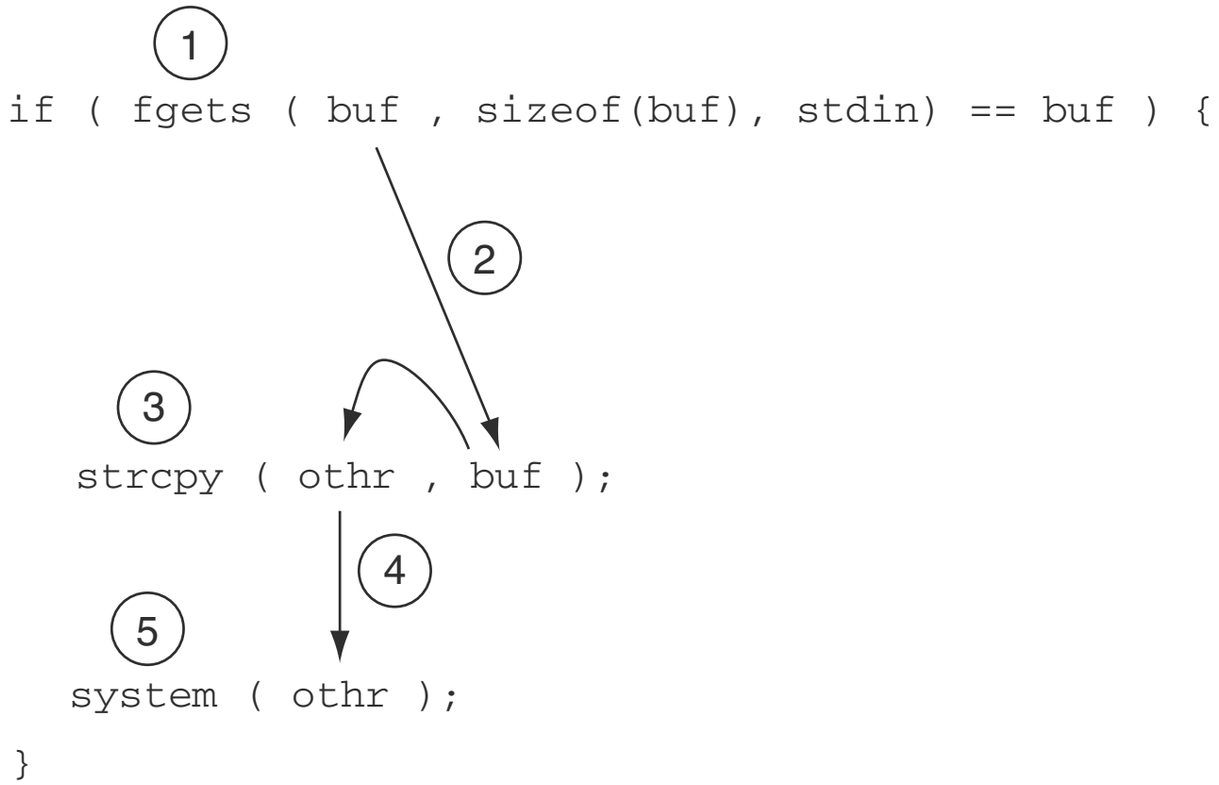
Example external rules

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

```
<DataflowSinkRule formatVersion="3.2" language="cpp">
  <MetaInfo><Group name="package">C Core</Group></MetaInfo>
  <RuleID>AA212456-92CD-48E0-A5D5-E74CC26A276F</RuleID>
  <VulnKingdom>Input Validation and Representation</VulnKingdom>
  <VulnCategory>Command Injection</VulnCategory>
  <DefaultSeverity>4.0</DefaultSeverity>
  <Description ref="desc.dataflow.cpp.command_injection"/>
  <Sink>
    <InArguments>0</InArguments>
    <Conditional>
      <Not>
          <TaintFlagSet taintFlag="VALIDATED_COMMAND_INJECTION"/>
      </Not>
    </Conditional>
  </Sink>
  <FunctionIdentifier>
     <FunctionName><Value>system</Value></FunctionName>
  </FunctionIdentifier>
</DataflowSinkRule>
```

- **RATS rule** (lhs): reports violation whenever 1st argument to **system()** is not a constant (RATS = Rough Auditing Tool for Security)
- **Fortify rule** (rhs): reports same violation but only if there is a path through the program where the attacker can control the 1st argument, and the argument has not been validated to prevent injections
  - Also classifies vulnerability, links to textual description, etc.

Making rules precise — taint analysis



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

- **Source rules** define locations where tainted data enters system
- **Sink rules** define locations that should not receive tainted data
- **Pass-through rules** define how a function manipulates tainted data

  E.g., if input is tainted, output is similarly (or differently) tainted
- **Clense rule**: special pass-through rules that remove taint

  E.g., validation functions

Taint analysis rules (cont.)

- Taint is just an attribute on data. In practice not binary
  - Input from **main()** may not be trustworthy, but guaranteed to be null terminated string
  - Data may have undergone certain checks
  - So represent taint by different taint flags
- Sinks may be dangerous only when data has a certain taint
- Rules can manipulate taint in either additive or subtractive manner
- All of this must be specified, also for user-defined functions
  - Alternative: resort to binary taint, but then too many false positives
  - Engineering effort is most of the work

Handling Input

Why, what, and how

Input validation — overview

- Security maxim: **Validate all input!**
- **Attack surface** = all places where program accepts input
  - All program entry points and function calls taking external input
  - Can use static analysis tools to help audit these!
- Attack surface includes:

  Command-line parameters, config. files, database data, environment variables, network services, registry values, temporary files
- All of these have been exploited in the past!

  (See Chess/West for dozens of examples)
- We will look at several examples and how scanning helps

Example: Database Access

```
ResultSet rs = stmt.executeQuery();
rs.next();
int balance = rs.getInt(1);
```

- The database is input too! Is above code OK?
  - Next positions cursor to first row
  - rs.getInt(1) gets value of first column, here the balance
- What if database is corrupted? Security/Functionally relevant!
  - Could be zero or multiple rows. This should be checked.
- Fix: validate input!

```
ResultSet rs = stmt.executeQuery();
if (!rs.next()) {
  throw new LookupException("no balance row");
}
if (!rs.isLast()) {
  throw new LookupException("more than one balance row");
}
int balance = rs.getInt(1);
```

Rule to find this problem

- Identify where program ignores number of rows a RuleSet contains
  - Find calls to RuleSet.next() that are call statements and therefore not in a predicate

```
FunctionCall fc:
  (fc.function is [name == "next" and
                   enclosingClass.supers contains
   [Class: name == "java.sql.ResultSet"]]) and
  (fc in [CallStatement:])
```

- **Question**: why wouldn't you trust your data base?
  - Don't trust anything! Stop compromise escalation!
  - But if almost everything is tainted, analysis should be precise

Example: Trust boundaries

```
status = request.getParameter("status");
if (status != null && status.length() > 0) {
   session.setAttribute("USER_STATUS", status);
```

- Trust boundaries: knowing what to trust for what
  - ✦ Taint analysis useful to track of how data flows between boundaries
  - ✦ Above Java accepts HTTP request parameter status and stores it in session object. May be sanitized for use within session, or not.
- Following JSP use to print value from above session object

```
<%
   String user_state = "Unknown";
   try {
      HttpSession user_session = Init.sessions.get(tmpUser.getUser());
      user_state = user_session == null ? "Unknown" :
                   (String)user_session.getAttribute("USER_STATUS");

   }
   ...
%>
<%=user_state %>
```

- Trust boundary crossed from input to output: c-x scripting attack!

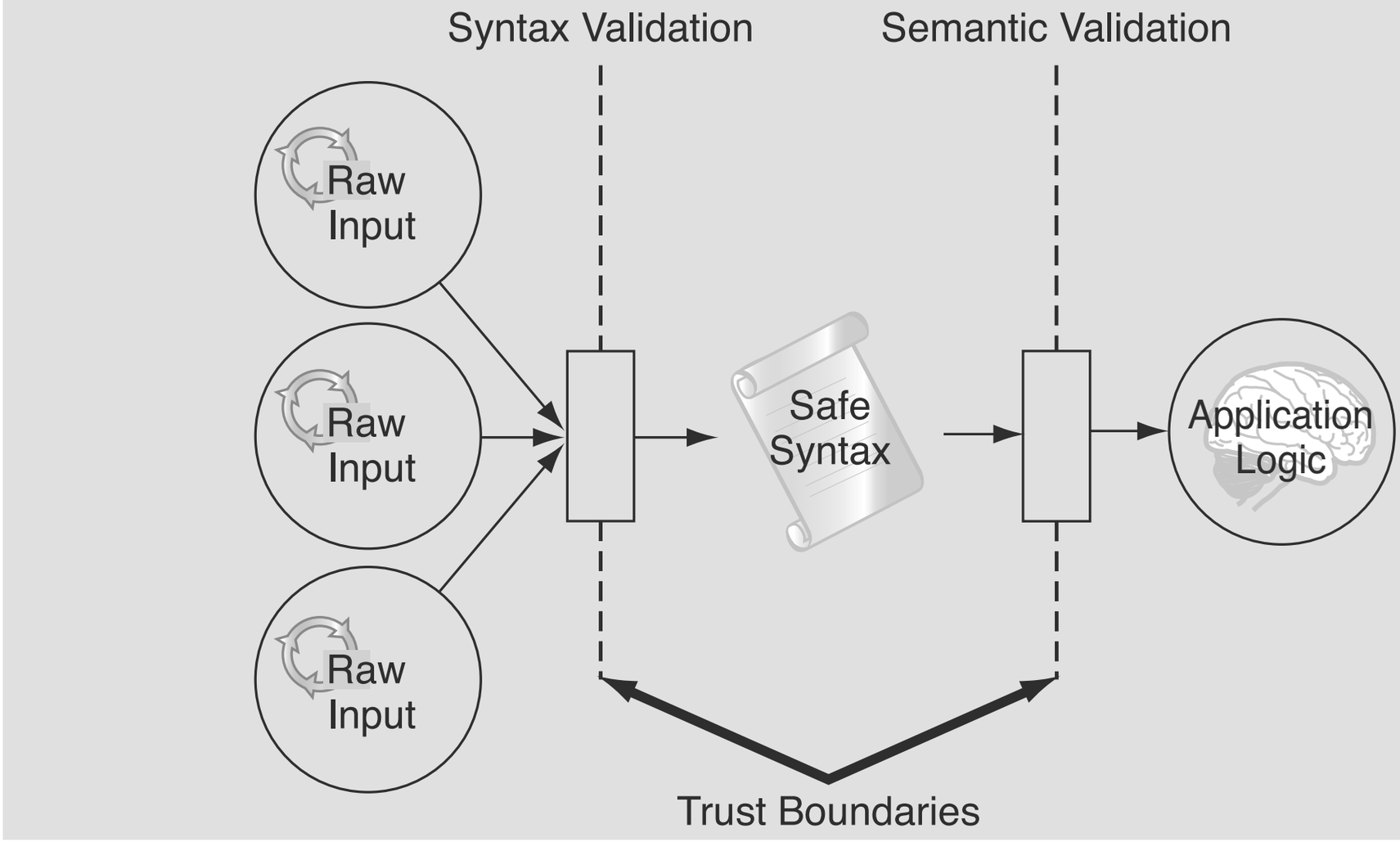
Trust boundary rules (high level)

Source rule:

```
Function: javax.servlet.http.HttpServletRequest.getParameter()
Postcondition: return value is tainted
```

Sink rule:

```
Function: javax.servlet.http.Session.setAttribute()
Precondition: arguments must not be tainted
```



SQL injection Java example

```
lastName = request.getParameter("last_name");
query = "SELECT phone FROM phnbk WHERE lnam = '"
        +lastName+"'";
rs = stmt.executeQuery(query);
```

- Detect using following 3 rules for taint propagation
  - **Source rule**

    Function: javax.servlet.http.HTTPServletRequest.getparamter()

    Postcondition: return value is tainted
  - **Pass-through rule**

    Function: string concatenation

    Postcondition: result is tainted if either input string is tainted
  - **Sink rule**

    Function: java.sql.Statement.executeQuery()

    Precondition: argument must not be tainted

- **Question**: how can you avoid SQL injections?

Conclusions

Hundreds of properties and rules

- Chess/West book is an excellent introduction to them
  - Primer on secure programming and security functionality
  - E.g., stack/heap overflows, safe language dialects, error handling, input validation frameworks, XML problems, safe storage of passwords, race conditions, …
  - Highly recommended reading!
- Name of the game stays the same: vulnerability patterns + rules
- Experience of different tool makers similar
  - Simple analysis combined with good rules yields dramatic results

But how simple is the analysis?

Findbugs experience

simple analysis combined with effective rules

Implementation strategies used by the detectors can be divided into several rough categories:

- **Class structure and inheritance hierarchy only. Some of the detectors simply look at the structure of the analyzed classes without looking at the code.**
- **Linear code scan. These detectors make a linear scan through the bytecode for the methods of analyzed classes, using the visited instructions to drive a state machine. These detectors do not make use of complete control flow information; however, heuristics (such as identifying the targets of branch instructions) can be effective in approximating control flow.**
- **Control sensitive. These detectors use an accurate control flow graph for analyzed methods.**
- **Dataflow. The most complicated detectors use dataflow analysis to take both control and data flow into account. An example is the null pointer dereference detector.**

None of the detectors make use of analysis techniques more sophisticated than what might be taught in an under-graduate compiler course.

Source: Hovemeyer, Pugh, Finding Bugs is Easy, OOPSOLA 2004

Simple analysis ≠ simple rules or concepts

Double-Checked Locking (DC) rule

- Consider “lazy initialization”, delaying initialization of owned object until it is really needed

```
class SomeClass {
  private Resource resource = null;
  public Resource getResource() {
    if (resource == null)
      resource = new Resource();
    return resource;
  }
```

OPTION 1: test and set

```
class SomeClass {
  private Resource resource = null;
  public Resource getResource() {
    if (resource == null) {
      synchronized {
        if (resource == null)
          resource = new Resource();
      }
    }
    return resource;
  }}
```

DCL variant for multi-threading

- Option 1 has a race condition
- Option 2: compilers, microprocessors, and caches can do reordering
  - It is possible thread A enters synchronized and calls **new Resource()**
  - Thread B enters **getResource()**
  - Due to reordering, B gets a non-null reference to a partially built resource!
- Its very subtle! Read the "Double-Checked Locking is Broken Declaration"

Evaluation (partial)

	classpath-0.08				rt.jar 1.5.0 build 59			
code	warnings	serious	mostly harmless	false pos	warnings	serious	mostly harmless	false pos
DC	1	100%	0%	0%	88	100%	0%	0%
EC	0	—	—	—	8	100%	0%	0%
IS2	46	58%	23%	17%	116	44%	47%	7%
NP	7	85%	0%	14%	37	100%	0%	0%
NS	0	—	—	—	12	25%	66%	8%
OS	6	50%	0%	50%	13	15%	0%	84%
RCN	5	80%	0%	20%	35	57%	0%	42%
RR	9	100%	0%	0%	12	91%	0%	8%
RV	5	100%	0%	0%	7	71%	0%	28%
UR	3	66%	0%	33%	4	100%	0%	0%
UW	2	0%	0%	100%	6	50%	0%	50%
Wa	3	0%	0%	100%	8	37%	0%	62%

	eclipse-3.0				drjava-stable-20040326			
code	warnings	serious	mostly harmless	false pos	warnings	serious	mostly harmless	false pos
DC	88	100%	0%	0%	0	—	—	—
EC	19	57%	0%	42%	0	—	—	—
IS2	63	61%	22%	15%	2	0%	0%	100%
NP	70	78%	7%	14%	0	—	—	—
NS	14	78%	21%	0%	0	—	—	—
OS	26	46%	0%	53%	4	100%	0%	0%
RCN	69	40%	11%	47%	0	—	—	—
RR	39	38%	0%	61%	0	—	—	—
RV	8	100%	0%	0%	0	—	—	—
UR	4	50%	50%	0%	1	0%	100%	0%
UW	7	28%	0%	71%	3	100%	0%	0%
Wa	12	25%	0%	75%	3	100%	0%	0%

Summary

- Code scanning should play a central role in code review
  - ✦ Aids understanding code
  - ✦ Helps finding common bugs
- Surprisingly effective!
  - ✦ Everyone makes dumb mistakes
  - ✦ Simple patterns can describe remarkably subtle bugs
  - ✦ Threads/crypto/… more complex than most people think
- Pragmatic conservative static analysis techniques play a major role
  - ✦ Augmented with lots of domain knowledge
- Topic is deep. We only scratched surface on analysis methods