Lecture 16

Pointer Analysis

1. Why is pointer analysis useful and hard?
2. Datalog
3. Context-insensitive, flow-insensitive pointer analysis
4. Context sensitivity

Readings: Chapter 12

Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams
John Whaley and Monica S. Lam
SQL Injection Errors

Hacker → Browser → Web App → Database

Give me Bob’s credit card #
Delete all records
SQL Injection Pattern

Dynamically:

```
User supplies text

o = req.getParameter();
stmt.executeQuery(o);
```

Text controls the database
ParameterParser.java:586
String session.ParameterParser.getRawParameter(String name)

```java
public String getRawParameter(String name) throws ParameterNotFoundException {
    String[] values = request.getParameterValues(name);
    if (values == null) {
        throw new ParameterNotFoundException(name + " not found");
    } else if (values[0].length() == 0) {
        throw new ParameterNotFoundException(name + " was empty");
    }
    return (values[0]);
}
```

ParameterParser.java:570
String session.ParameterParser.getRawParameter(String name, String def)

```java
public String getRawParameter(String name, String def) {
    try {
        return getRawParameter(name);
    } catch (Exception e) {
        return def;
    }
}
```
In Practice (II)

ChallengeScreen.java:194
Element lessons.ChallengeScreen.doStage2(WebSession s)

```java
String user = s.getParser().getRawParameter(USER, "");
StringBuffer tmp = new StringBuffer();
tmp.append("SELECT cc_type, cc_number from user_data WHERE userid = "");
tmp.append(user);
tmp.append(""");
tmp.append(""");
query = tmp.toString();
Vector v = new Vector();
try {
    ResultSet results = statement3.executeQuery(query);
    ...
```
Vulnerabilities in Web Applications

Inject
- Parameters
- Hidden fields
- Headers
- Cookie poisoning

Exploit
- SQL injection
- Cross-site scripting
- HTTP splitting
- Path traversal

Lam & Whaley
CS243: Pointer Analysis
Key: Information Flow
PQL: Program Query Language

- PQL: Originally developed to find security errors by monitoring run-time behavior

- Query on the dynamic behavior based on object entities

- Example of a PQL language

  ```
  o = req.getParameter();
  stmt.executeQuery(o);
  ```
Dynamic vs. Static Pattern

Dynamically:

```java
o = req.getParameter();
stmt.executeQuery(o);
```

Statically:

```java
p1 = req.getParameter();
stmt.executeQuery(p2);
```

$p_1$ and $p_2$ point to same object?
Pointer alias analysis
Today’s Security Analyses

• 2 kinds of analysis
  – Conservative
    • All errors are reported: program is certified to have no security errors
    • Include: false positives
  – Opportunistic
    • Only a subset of errors is reported
    • Include: false positives and false negatives

Quiz: Why are most analyses opportunistic?
Can We Apply Dataflow Analysis to Pointer Alias Analysis?

• What are the challenges?
What’s So Hard about Pointer Alias Analysis?

• Choice of abstraction
  – Naming: an unbounded number of dynamically allocated objects
  – Aliases vs. points-to analysis

• Representation
  – Needs to model each field in an object
  – The state of the information is large especially if flow-sensitive

• Precision
  – C, C++ uses address arithmetic.
    • Every pointer can be written to if the address is unknown.
  – A write through an unknown pointer pollutes many pointer variables.
    • Worse if it is not a typed language
  – A read through an unknown pointer gets all possible values (incl. pointers)
  – Needs whole-program interprocedural analysis
    • Must model parameter passing
    • A callee’s side effects depend on the caller’s context (applies transitively)
  – Imprecision will propagate many points-to relationships
    • The size of the state grows with imprecision
Automatic Conservative Analysis Generation

Programmer:
Security analysis
in 10 lines

Compiler Writer:
Ptr analysis in 10 lines

PQL

Datalog

bddbddb
(BDD-based
deductive database)
with
Active Machine Learning

1000s of lines
1 year tuning

BDD operations

BDD (Binary Decision Diagrams): 10,000s-lines library

Compiler Writer: 10,000s-lines library

Lam & Whaley

CS243: Pointer Analysis
Goals of the Lecture

• Pointer analysis
  – Interprocedural, context-sensitive, flow-insensitive
    (Dataflow: intraprocedural, flow-sensitive)

• Power of languages and abstractions

• Elegant abstractions
  – Datalog: A deductive database
    (A database that can make deductions from stored data)
  – BDDs: Binary decision diagrams
    (Most cited CS papers for many years)
Outline

Pointer Analysis

1. Why is pointer analysis useful and hard?
2. Datalog
3. Context-insensitive, flow-insensitive pointer analysis
4. Context sensitivity
2. Datalog: a Deductive Database

• Relations as predicates
  – \( p(X_1, X_2, \ldots X_n) \)
    • \( X_1, X_2, \ldots X_n \) are variables or constants

• Database operations: logical rules
  – With recursion

• Unified syntax
  – Raw data: Extensional database (EDB)
  – Deduced results: Intensional database (IDB)
Example: Call graph edges

Predicate vs. Relation

- **Predicates**
  - \( \text{calls}(x,y) : x \text{ calls } y \) is true
  - Ground atoms: predicates with constant arguments

- **Relations**
  - \( \text{calls}(A,B) \)
  - \( \text{calls}(A,C) \)
  - \( \text{calls}(A,D) \)
  - \( \text{calls}(B,D) \)
  - \( \text{calls}(C,D) \)
  - \( \text{calls}(x,y) : x, y \) is in a “calls” relationship
  - Extensional database: tuples representing facts
Datalog Programs:
Set of Rules (Intensional DB)

• $H : - B_1, B_2, \ldots, B_n$.
• LHS is true if RHS is true
  – Rules define the intensional database
• Example: Datalog program to compute call*
  – transitive closure of calls relation
  – $\text{calls}^*(x, y)$ if $x$ calls $y$ directly or indirectly
  – $\text{calls}^*(x, y) : - \text{calls}(x, y)$
  – $\text{calls}^*(x, z) : - \text{calls}^*(x, y), \text{calls}^*(y, z)$.
• Result:
  – set of ground atoms inferred by applying the rules until no new inferences can be made
**Datalog vs. SQL**

- **SQL**
  - Imperative programming:
    - join, union, projection, selection
  - Explicit iteration
- **Datalog: logical database language**
  - Declarative programming
  - Recursive definition: fixpoint computation
  - Negation is not monotone
    - Can lead to oscillation in fix-point calculation
  - Stratified: separates rules into groups
    - Compute one group at a time
    - Can negate only the results from previous strata
Why use a Deductive Database for Pointer Analysis?

- Pointer analysis produces “intermediate” results to be consumed in analysis.
- Allow queries of specific subsets of results
- Results of queries can be further queried in a uniform way
Outline

Pointer Analysis

1. Motivation: security analysis
2. Datalog
3. Context-insensitive, flow-insensitive pointer analysis
4. Context sensitivity
3. Flow-Insensitive Points-to Analysis

• Alias analysis:
  – Can two pointers point to the same location?
  – *a, *(a+8)

• Points-to analysis:
  – What objects does each pointer points to?
  – Two pointers cannot be aliased
    if they must point to different objects
How to Name Objects?

- Objects are dynamically allocated
- Use finite names to refer to unbounded objects
- 1 scheme: Name an object by its allocation site

```c
main () {
    f() {
        p = f(); A: a = new O();
        q = f(); B: b = new O();
    }
    return a;
}
```

- If constructors are used
  - name an object by the call site to the constructor.
Points-To Analysis for Java

- Variables ($v \in V$):
  - local variables in the program

- Heap-allocated objects ($h \in H$)
  - has a set of fields ($f \in F$)
  - named by allocation site

- Points-to analysis: to compute predicates
  - $vP(v, h)$: variable $v$ can point to object $h$
  - $hP(h_1, f, h_2)$: object $h_1$ field $f$ can point to object $h_2$
Program Abstraction

- Allocations
  \[ h: v = \text{new} \ c \]
- Store
  \[ v_1.f = v_2 \]
- Loads
  \[ v_2 = v_1.f \]
- Moves, arguments:
  \[ v_1 = v_2 \]
- Assume: a (conservative) call graph is known a priori for now
  - Call:
    \[ \text{formal} = \text{actual} \]
  - Return:
    \[ \text{actual} = \text{return value} \]
Pointer Analysis Rules

Object creation
\( \text{vP}(v, h) ::= \text{“h: T v = new T()”}. \)

Assignment
\( \text{vP}(v_1, h_1) ::= \text{“v}_1 = v_2”, \text{vP}(v_2, h_1). \)

Store
\( \text{hP}(h_1, f, h_2) ::= \text{“v}_1.f = v_2”, \text{vP}(v_1, h_1), \text{vP}(v_2, h_2). \)

Load
\( \text{vP}(v_2, h_2) ::= \text{“v}_2 = v_1.f”, \text{vP}(v_1, h_1), \text{hP}(h_1, f, h_2). \)
Flow-Insensitive, Context-Insensitive Pointer Alias Analysis

• Specified by a few Datalog rules
  – Creation sites
  – Assignments
  – Stores
  – Loads
• Apply rules until they converge
Example

```c
void main() {
    x = new C();  (main@1)
    y = new C();  (main@2)
    z = new C();  (main@3)
    m(x,y);
    n(z,x);
    q = z.f;
}

void m(C a, C b) {
    n(a,b);
}

void n(C c, C d) {
    c.f = d;
}
```

1. what are the objects pointing to? (perfect case)

![Diagram of object pointers]

2. What does this algorithm produce? (flow-insensitive, context-insensitive)

![Diagram of object pointers]
Pointer Analysis in Datalog

Domains

\( V = \) variables
\( H = \) heap objects
\( F = \) fields

Extensional database EDB (input) relations

\[ vP_0 (v:V, h:H) \quad : \text{object allocation sites} \]
\[ \text{assign} (v_1:V, v_2:V) \quad : \text{assignment instructions} \ (v_1 = v_2;) \text{ and parameter passing} \]
\[ \text{store} (v_1:V, f:F, v_2:V) \quad : \text{store instructions} \ (v_1.f = v_2;) \]
\[ \text{load} (v_1:V, f:F, v_2:V) \quad : \text{load instructions} \ (v_2 = v_1.f;) \]

Intensional database IDB (computed) relations

\[ vP(v:V, h:H) \quad : \text{variable points-to relation} \ (\text{variable} \ v \ \text{can point to object} \ h) \]
\[ hP(h_1:H, f:F, h_2:H) \quad : \text{heap points-to relation} \ (\text{object} \ h_1 \ \text{field} \ f \ \text{can point to} \ h_2) \]

Rules

\[ vP(v, h) \quad :- \ vP_0(v, h). \]
\[ vP(v_1, h) \quad :- \ \text{assign}(v_1, v_2), \ vP(v_2, h). \]
\[ hP(h_1, f, h_2) \quad :- \ \text{store}(v_1, f, v_2), \ vP(v_1, h_1), \ vP(v_2, h_2). \]
\[ vP(v_2, h_2) \quad :- \ \text{load}(v_1, f, v_2), \ vP(v_1, h_1), \ hP(h_1, f, h_2). \]
Step 1: Assign numbers to elements in domain

```c
void main() {
    x = new C();
    y = new C();
    z = new C();
    m(x,y);
    n(z,x);
    q = z.f;
}
void m(C a, C b) {
    n(a,b);
}
void n(C c, C d) {
    c.f = d;
}
```

Domains

<table>
<thead>
<tr>
<th>V</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>'x' : 0</td>
<td>'main@1' : 0</td>
</tr>
<tr>
<td>'y' : 1</td>
<td>'main@2' : 1</td>
</tr>
<tr>
<td>'z' : 2</td>
<td>'main@3' : 2</td>
</tr>
<tr>
<td>'a' : 3</td>
<td></td>
</tr>
<tr>
<td>'b' : 4</td>
<td></td>
</tr>
<tr>
<td>'c' : 5</td>
<td></td>
</tr>
<tr>
<td>'d' : 6</td>
<td></td>
</tr>
<tr>
<td>'q' : 7</td>
<td></td>
</tr>
</tbody>
</table>

Lam & Whaley
CS243: Pointer Analysis 30
void main() {
    x = new C();
    y = new C();
    z = new C();
    m(x,y);
    n(z,x);
    q = z.f;
}

void m(C a, C b) {
    n(a,b);
}

void n(C c, C d) {
    c.f = d;
}
Step 3: Generate Predicate Dependency Graph

Rules

\[
\begin{align*}
vP(v, h) & \;:\; vP_0(v, h). \\
vP(v_1, h) & \;:\; assign(v_1, v_2), \; vP(v_2, h). \\
hP(h_1, f, h_2) & \;:\; store(v_1, f, v_2), \; vP(v_1, h_1), \; vP(v_2, h_2). \\
vP(v_2, h_2) & \;:\; load(v_1, f, v_2), \; vP(v_1, h_1), \; hP(h_1, f, h_2).
\end{align*}
\]
Step 4: Determine Iteration Order

In topological sort of strongly connected components
Self-cycles before bigger cycles
Step 5: Apply rules until convergence

Rules

\[ vP(v,h) :: vP_0(v,h). \]
\[ vP(v_1,h) :: assign(v_1,v_2), vP(v_2,h). \]
\[ hP(h_1,f,h_2) :: store(v_1,f,v_2), vP(v_1,h_1), vP(v_2,h_2). \]
\[ vP(v_2,h_2) :: load(v_1,f,v_2), vP(v_1,h_1), hP(h_1,f,h_2). \]

Relations

\[ vP_0 \]
\[ assign \]
\[ vP \]
\[ hP \]

\[ vP_0('x','main@1'). \]
\[ assign('a','x'). \]
\[ vP_0('y','main@2'). \]
\[ assign('b','y'). \]
\[ vP_0('z','main@3'). \]
\[ assign('c','z'). \]
\[ store \]
\[ store('c','f','d'). \]
\[ load \]
\[ load('z','f','q'). \]
Step 5: Apply rules until convergence

Rules
\[ vP(v,h) \leftarrow vP_0(v,h). \]
\[ vP(v_1,h) \leftarrow assign(v_1,v_2), vP(v_2,h). \]
\[ hP(h_1,f,h_2) \leftarrow store(v_1,f,v_2), vP(v_1,h_1), vP(v_2,h_2). \]
\[ vP(v_2,h_2) \leftarrow load(v_1,f,v_2), vP(v_1,h_1), hP(h_1,f,h_2). \]

Relations
\[ vP_0 \]
\[ vP_0('x','main@1'). \]
\[ vP_0('y','main@2'). \]
\[ vP_0('z','main@3'). \]

assign
\[ assign('a','x'). vP('x','main@1'). \]
\[ assign('b','y'). vP('y','main@2'). \]
\[ assign('c','z'). vP('z','main@3'). \]
\[ assign('d','x'). \]
\[ assign('c','a'). \]
\[ assign('d','b'). \]

load
\[ load('z','f','q'). \]

void main() {
    x = new C();
    y = new C();
    z = new C();
    m(x,y);
    void n(C c, C d) {
        c.f = d;
        n(z,x);
    }
    q = z.f;
}
Step 5: Apply rules until convergence

Rules
\[ vP(v,h) := vP_0(v,h). \]
\[ vP(v_1,h) := \text{assign}(v_1,v_2), vP(v_2,h). \]
\[ hP(h_1,f,h_2) := \text{store}(v_1,f,v_2), vP(v_1,h_1), vP(v_2,h_2). \]
\[ vP(v_2,h_2) := \text{load}(v_1,f,v_2), vP(v_1,h_1), hP(h_1,f,h_2). \]

Relations
\[ vP_0 \]
\[ vP_0('x','main@1'). \]
\[ vP_0('y','main@2'). \]
\[ vP_0('z','main@3'). \]

assign
\[ \text{assign('a','x')} \]
\[ \text{assign('b','y')} \]
\[ \text{assign('c','z')} \]
\[ \text{assign('d','x')} \]
\[ \text{assign('c','a')} \]
\[ \text{assign('d','b')} \]

vP
\[ \text{vP('x','main@1')} \]
\[ \text{vP('y','main@2')} \]
\[ \text{vP('z','main@3')} \]
\[ \text{vP('a','main@1')} \]
\[ \text{vP('d','main@1')} \]
\[ \text{vP('b','main@2')} \]
\[ \text{vP('c','main@3')} \]

hP
\[ \text{store('c','f','d')} \]

load
\[ \text{load('z','f','q')} \]

void main() {  void m(C a, C b) {
    x = new C();  n(a,b);
    y = new C();
    z = new C();
    m(x,y);
    n(z,x);
    q = z.f;
    }
}
Step 5: Apply rules until convergence

Rules

\[ vP(v, h) :\]
\[ vP(0, v, h) : assign(v, v', vP(v', v), vP(v', vP(v, h)). \]
\[ hP(h_1, f, h_2) : store(v, v', vP(v, h), vP(v, vP(h, h)), \]
\[ vP(0, v, h) : load(v, f, v', vP(v, h), \]
\[ vP(0, v, h) : \]

Relations

\[ vP_0 \]
\[ assign \]
\[ vP \]
\[ hP \]

void main() {
    x = new C();
    y = new C();
    z = new C();
    m(x, y);
    n(z, x);
    q = z.f;
}

void m(C a, C b) {
    n(a, b);
}

void n(C c, C d) {
    c.f = d;
}

Lam & Whaley  
CS243: Pointer Analysis  
37

store

store('c', 'f', 'd').

load

load('z', 'f', 'q').

(red text: imprecision)
Step 5: Apply rules until convergence

Rules

\[
\begin{align*}
\text{vP}(v,h) & :& \text{vP}_0(v,h). \\
\text{vP}(v_1,h) & :& \text{assign}(v_1,v_2), \text{vP}(v_2,h). \\
\text{hP}(h_1,f,h_2) & :& \text{store}(v_1,f,v_2), \text{vP}(v_1,h_1), \text{vP}(v_2,h_2). \\
\text{vP}(v_2,h_2) & :& \text{load}(v_1,f,v_2), \text{vP}(v_1,h_1), \text{hP}(h_1,f,h_2).
\end{align*}
\]

Relations

\[
\begin{align*}
\text{vP}_0 & : \\
\text{vP}_0('x','\text{main@1}') & : \\
\text{vP}_0('y','\text{main@2}') & : \\
\text{vP}_0('z','\text{main@3}').
\end{align*}
\]

\[
\begin{align*}
\text{store} & : \\
\text{store('c','f','d').}
\end{align*}
\]

\[
\begin{align*}
\text{load} & : \\
\text{load('z','f','q').}
\end{align*}
\]

\[
\begin{align*}
\text{assign} & : \\
\text{assign('a','x').} & : \text{vP('x','\text{main@1}').} \\
\text{assign('b','y').} & : \text{vP('y','\text{main@2}').} \\
\text{assign('c','z').} & : \text{vP('z','\text{main@3}').} \\
\text{assign('d','x').} & : \text{vP('a','\text{main@1}').} \\
\text{assign('c','a').} & : \text{vP('d','\text{main@1}').} \\
\text{assign('d','b').} & : \text{vP('b','\text{main@2}').}
\end{align*}
\]

\[
\begin{align*}
\text{hP} & : \\
\text{hP('main@1','f','main@1').} & : \\
\text{hP('main@1','f','main@2').} & : \\
\text{hP('main@3','f','main@1').} & : \\
\text{hP('main@3','f','main@2').}
\end{align*}
\]

Lam & Whaley

CS243: Pointer Analysis

38
Step 5: Apply rules until convergence

Rules

\[ \text{vP}(v,h) :: \text{vP}_0(v,h) \]
\[ \text{vP}(v_1,h) :: \text{assign}(v_1,v_2), \text{vP}(v_2,h) \]
\[ \text{hP}(h_1,f,h_2) :: \text{store}(v_1,f,v_2), \text{vP}(v_1,h_1), \text{vP}(v_2,h_2) \]
\[ \text{vP}(v_2,h_2) :: \text{load}(v_1,f,v_2), \text{vP}(v_1,h_1), \text{hP}(h_1,f,h_2) \]

Relations

\text{vP}_0

\[ \text{vP}_0('x','\text{main@1}') \]
\[ \text{vP}_0('y','\text{main@2}') \]
\[ \text{vP}_0('z','\text{main@3}') \]

\text{assign}

\[ \text{assign}('a','x') \]
\[ \text{assign}('b','y') \]
\[ \text{assign}('c','z') \]
\[ \text{assign}('d','x') \]
\[ \text{assign}('c','a') \]
\[ \text{assign}('d','b') \]

\text{vP}

\[ \text{vP}('x','\text{main@1}') \]
\[ \text{vP}('y','\text{main@2}') \]
\[ \text{vP}('z','\text{main@3}') \]
\[ \text{vP}('a','\text{main@1}') \]
\[ \text{vP}('d','\text{main@1}') \]
\[ \text{vP}('b','\text{main@2}') \]
\[ \text{vP}('c','\text{main@3}') \]
\[ \text{vP}('c','\text{main@1}') \]
\[ \text{vP}('d','\text{main@2}') \]
\[ \text{vP}('q','\text{main@1}') \]
\[ \text{vP}('q','\text{main@2}') \]

\text{hP}

\[ \text{hP}('\text{main@1}','f','\text{main@1}') \]
\[ \text{hP}('\text{main@1}','f','\text{main@2}') \]
\[ \text{hP}('\text{main@3}','f','\text{main@1}') \]
\[ \text{hP}('\text{main@3}','f','\text{main@2}') \]

(void m(C a, C b) {
    x = new C();
    y = new C();
    z = new C();
    m(x,y);
}

(void n(C c, C d) {
    c.f = d;
}

void main() {
    x = new C();
    y = new C();
    z = new C();
    m(x,y);
    n(z,x);
    q = z.f;
}

(store 'c','f','d').

(assign 'a','x').
(assign 'b','y').
(assign 'c','z').
(assign 'd','x').
(assign 'c','a').
(assign 'd','b').

Lam & Whaley
Call Graphs

- Previous algorithm assumes an a priori call graph
  - Every possible method is assumed to be invoked
- Virtual method invocation is determined by the type hierarchy

```java
void draw(shape s) {
    int i = s.lines();
    ...
}
Square s;
cha(square, lines, lines_{rectangle})
```

- Class hierarchy analysis: `cha(t, n, m)`
  - Given an invocation `v.n(...)`, if `v` points to object of type `t`, then `m` is the method invoked
  - `m` belongs to the first superclass that defines `n`
Use Pointer Analysis Result to Reduce the Call Graph

• Instead of a priori call graph, determine on the fly:
  – Discover **points-to results**
    determine the types,
    use **CHA** to find methods called
• Create extensional database from the program
  hType(h, t): h has type t
  invokes(s, m): statement s calls method m

  invokes(s, m) :- "s: v.n(...)", vP(v, h), hType (h, t), cha(t, n, m).

• Parameter passing:
  actual(s, i, v) : v is the ith actual parameter in call site s.
  formal(m, i, v) : v is the ith formal parameter declared in method m.

  vP(v, h):- invokes(s, m), formal(m, i, v), actual(s, i, w), vP(w, h).
Outline

Pointer Analysis

1. Motivation: security analysis
2. Datalog
3. Context-insensitive, flow-insensitive pointer analysis
4. Context sensitivity
4. Context-Sensitive Pointer Analysis

L1: a=malloc();
    a=id(a);

id(x)
{return x;}

L2: b=malloc();
b=id(b);

context-sensitive

context-insensitive
Even without recursion, # of contexts is exponential!
Recursion
Top 20 Sourceforge Java Apps

Number of Clones

Size of program (variable nodes)

Number of clones

10^0

10^4

10^8

10^12

10^16

1000

10000

100000

1000000
Cloning-Based Algorithm

• Apply the context-insensitive algorithm to the program to discover the call graph

• Context-sensitive analysis
  – Find strongly connected components
  – Create a “clone” for every context
  – Apply the context-insensitive algorithm to cloned call graph

• How to handle the exponential growth
  – Lots of redundancy in result
  – Exploit redundancy by clever use of BDDs (binary decision diagrams)
Binary Decision Diagram (BDD)

- BDD: Binary Decision Diagrams
  - Designed to exploit similarities in an exponential number of states
  - Usage: logic synthesis, verification

- For context sensitive pointer analysis:
  - Implement Datalog with BDDs
  - Relations are expressed as binary decision diagrams
  - Using BDDs effectively takes 1 year of tuning from not finishing to within minutes (old machine)

- BDDBDDB (Binary Decision Diagram-Based Deductive Database)
  - Datalog rules are implemented with BDD operators
Experimental Results

• Top 20 Java projects on SourceForge
  – Real programs with 100K+ users each

• Using automatic bddbddb solver
  – Each analysis only a few lines of code
  – Easy to try new algorithms, new queries

• Test system:
  – Pentium 4 2.2GHz, 1GB RAM
  – RedHat Fedora Core 1, JDK 1.4.2_04, javabdd library, Joeq compiler
Analysis time

y = 0.0078x^{2.3233}

R^2 = 0.9197
Analysis memory

$y = 0.3609x^{1.4204}$

$R^2 = 0.8859$
Benchmark

Nine large, widely used applications
- Blogging/bulletin board applications
- Used at a variety of sites
- Open-source Java J2EE apps
- Available from SourceForge.net
## Vulnerabilities Found

<table>
<thead>
<tr>
<th></th>
<th>SQL injection</th>
<th>HTTP splitting</th>
<th>Cross-site scripting</th>
<th>Path traversal</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Parameter</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>Cookie</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Non-Web</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>9</strong></td>
<td><strong>11</strong></td>
<td><strong>4</strong></td>
<td><strong>5</strong></td>
<td><strong>29</strong></td>
</tr>
<tr>
<td>Benchmark</td>
<td>Classes</td>
<td>Context insensitive</td>
<td>Context sensitive</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>---------</td>
<td>---------------------</td>
<td>-------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>jboard</td>
<td>264</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>blueblog</td>
<td>306</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>webgoat</td>
<td>349</td>
<td>51</td>
<td>6</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>blojsom</td>
<td>428</td>
<td>48</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>personalblog</td>
<td>611</td>
<td>460</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>snipsnap</td>
<td>653</td>
<td>732</td>
<td>27</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>road2hibernate</td>
<td>867</td>
<td>18</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>pebble</td>
<td>889</td>
<td>427</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>roller</td>
<td>989</td>
<td>378</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5356</strong></td>
<td><strong>2115</strong></td>
<td><strong>41</strong></td>
<td><strong>12</strong></td>
<td></td>
</tr>
</tbody>
</table>
Automatic Conservative Analysis Generation

Programmer: Security analysis in 10 lines

Compiler Writer: Flow-insensitive Context-sensitive Ptr analysis in 10 lines

PQL

Datalog

**bddd**

(BDD-based deductive database) with Active Machine Learning

BDD operations

BDD (Binary Decision Diagrams): 10,000s-lines library