Lecture 12
Pointer Analysis

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

Readings: Chapter 12

Interprocedural Pointer Analysis

Object creation
$$pts(v, h) : \text{"h: T v = new T()".}$$

Assignment
$$pts(v_1, h_1) : \text{"v_1 = v_2" & pts(v_2, h_1).}$$

Store
$$hpts(h_1, f, h_2) : \text{"v_1.f = v_2" & pts(v_1, h_1) & pts(v_2, h_2).}$$

Load
$$pts(v_2, h_2) : \text{"v_2 = v_1.f" & pts(v_1, h_1) & hpts(h_1, f, h_2).}$$

Parameter passing with virtual methods
$$\text{invokes (s, m) : \text{"s: v.n (...)" & pts(v,h) & hType (h,t) & cha (t,n,m) }$$
$$pts(v, h) : \text{- invokes (s, m) & }$$
$$\text{ formal (m, i, v) & actual (s, i, w) & pts (w, h) }$$
1. Web Applications

Hacker \(\leftrightarrow\) Browser \(\leftrightarrow\) Web App \(\leftrightarrow\) Database

Evil Input \(\downarrow\) Confidential information leak

SQL Injection Errors

Hacker \(\leftrightarrow\) Browser \(\leftrightarrow\) Web App \(\leftrightarrow\) Database

Give me Bob's credit card #
Delete all records
Happy-go-lucky SQL Query

User supplies: name, password

Java program:
String query =
“SELECT UserID, Creditcard FROM CCRec
WHERE Name = ”
+ name + “ AND PW = ”
+ password

Fun with SQL

“—”: “the rest are comments” in Oracle SQL
SELECT UserID, CreditCard FROM CCRec
WHERE:
Name = bob AND PW = foo
Name = bob— AND PW = x
Name = bob or 1=1— AND PW = x
Name = bob; DROP CCRec— AND PW = x
A Simple SQL Injection Pattern

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

In Practice

```java
public String getRawParameter(String name) throws ParameterNotFoundException {
    String[] values = requestgetParameterValues(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]);
}
```

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

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

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('');
query = tmp.toString();
Vector v = new Vector();
try
|
| ResultSet results = statement3.executeQuery(query);
| ...
```

Vulnerabilities in Web Applications

<table>
<thead>
<tr>
<th>Inject</th>
<th>Exploit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>SQL injection</td>
</tr>
<tr>
<td>Hidden fields</td>
<td>Cross-site scripting</td>
</tr>
<tr>
<td>Headers</td>
<td>HTTP splitting</td>
</tr>
<tr>
<td>Cookie poisoning</td>
<td>Path traversal</td>
</tr>
</tbody>
</table>

Advanced Compilers
M. Lam & J. Whaley
Key: Information Flow

PQL: Program Query Language

\[
o = \text{req.getParameter ( )};
\]
\[
\text{stmt.executeQuery ( o );}
\]

- Query on the dynamic behavior based on object entities
- Abstracting away information flow
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

Security Analysis

- **Classifications**
  - Conservative
    - All errors are reported
    - Include: false positives
  - Opportunistic
    - Only a subset of errors is reported
    - Include: false positives and false negatives

- **Pointer alias analysis**
  - No pointers
  - Flow-sensitive analysis?
  - Context-sensitive 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. Motivation: security analysis
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, … X_n)
    - X_1, X_2, … X_n are variables or constants
- Database operations: logical rules
  - With recursion
- Unified syntax
  - Raw data: Intensional database
  - Deduced results: Extensional database
Example: Call graph edges

Predicate vs. Relation

- **Predicates**
  - Calls (x, y): x calls y is true
  - Ground atoms: predicates with constant arguments

- **Relations**
  - Calls (x, y): x, y is in a "calls" relationship
  - Extensional database: tuples representing facts

```
calls(A,B)
calls(A,C)
calls(A,D)
calls(B,D)
calls(C,D)
```

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
  - calls*(x, y) if x calls y directly or indirectly
  - calls* (x, y) :- calls (x, y)
  - calls* (x, z) :- calls* (x, y) & 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 can lead to oscillation
  - Stratified: separates rules into groups
    - Compute one group at a time
    - Can negate only the results from previous strata

Datalog vs. Prolog

- Syntactically a subset of Prolog
- No function variables e.g. b in a(b(x,y), c)
- Truly declarative:
  - Rule ordering does not affect program semantics

- Bottom-up evaluation
  - Stratified Datalog always terminates on a finite database
Why use a Deductive Database for Pointer Analysis?

- Pointer analysis produces “intermediate” results to be consumed in analysis.
- Allow query of specific subsets of results
- Analysis as queries
- 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

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

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
  - Call:
    \[ \text{formal} = \text{actual} \]
  - Return:
    \[ \text{actual} = \text{return value} \]
Pointer Analysis Rules

Object creation
$$\text{pts}(v, h) := \text{"h: } T \ v = \text{new } T().".$$  

Assignment
$$\text{pts}(v_1, h_1) := \text{"} v_1 = v_2 \text{" } \& \text{ pts}(v_2, h_1).$$

Store
$$\text{hpts}(h_1, f, h_2) := \text{"} v_1.f = v_2 \text{" } \& \text{ pts}(v_1, h_1) \& \text{ pts}(v_2, h_2).$$

Load
$$\text{pts}(v_2, h_2) := \text{"} v_2 = v_1.f \text{" } \& \text{ pts}(v_1, h_1) \& \text{ hpts}(h_1, f, h_2).$$

Pointer Alias Analysis

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

```plaintext
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;
}
```

Pointer Analysis in Datalog

**Domains**
- \( V \) = variables
- \( H \) = heap objects
- \( F \) = fields

**EDB (input) relations**
- \( vP_0 \) : \((v : V, h : H)\) = object allocation sites
- \( assign(v_1 : V, v_2 : V)\) : assignment instructions \((v_1 = v_2)\) and parameter passing
- \( store(v_1 : V, f : F, v_2 : V)\) : store instructions \((v_1.f = v_2)\)
- \( load(v_1 : V, f : F, v_2 : V)\) : load instructions \((v_2 = v_1.f)\)

**IDB (computed) relations**
- \( vP \) : \((v : V, h : H)\) = variable points-to relation \((v\) can point to object \(h)\)
- \( hP \) : \((h_1 : H, f : F, h_2 : H)\) = heap points-to relation \((h_1\) field \(f\) can point to \(h_2)\)

**Rules**
- \( vP(v, h) \leftarrow vP_0(v, h) \)
- \( vP(v_1, h) \leftarrow assign(v_1, v_2), vP(v_2, h) \)
- \( hP(h_1, f, h_2) \leftarrow store(v_1, f, v_2), vP(v_1, h_1), vP(v_2, h_2) \)
- \( vP(v_2, h_2) \leftarrow load(v_1, f, v_2), vP(v_1, h_1), hP(h_1, f, h_2) \)
Step 1: Assign numbers to elements in domain

```java
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</td>
<td>'main@1': 0</td>
</tr>
<tr>
<td>y</td>
<td>'main@2': 1</td>
</tr>
<tr>
<td>z</td>
<td>'main@3': 2</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
</tr>
<tr>
<td>d</td>
<td>6</td>
</tr>
</tbody>
</table>

Step 2: Extract initial relations (EDB) from program

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

vP_0('x', 'main@1').
vP_0('y', 'main@2').
vP_0('z', 'main@3').
assign('a', 'x').
assign('b', 'y').
assign('c', 'z').
assign('d', 'x').
load('z', 'f', 'q').
assign('c', 'a').
assign('d', 'b').
store('c', 'f', 'd').
Step 3: Generate Predicate Dependency Graph

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

Step 4: Determine Iteration Order
Step 5: Apply rules until convergence

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

Relations
\[ vP_0 \]
\[ \text{assign} \]
\[ \text{vP} \]
\[ \text{hP} \]
\[ \text{store} \]
\[ \text{load} \]

Advanced Compilers
M. Lam & J. Whaley
Step 5: Apply rules until convergence

Rules

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

Relations

<table>
<thead>
<tr>
<th>assign</th>
<th>vP</th>
<th>hP</th>
</tr>
</thead>
<tbody>
<tr>
<td>vP_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vP_0('x','main@1')</td>
<td>assign('a','x')</td>
<td>vP('x','main@1')</td>
</tr>
<tr>
<td>vP_0('y','main@2')</td>
<td>assign('b','y')</td>
<td>vP('y','main@2')</td>
</tr>
<tr>
<td>vP_0('z','main@3')</td>
<td>assign('c','z')</td>
<td>vP('z','main@3')</td>
</tr>
<tr>
<td>store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>store('c','f','d')</td>
<td>assign('c','a')</td>
<td>vP('d','main@1')</td>
</tr>
<tr>
<td></td>
<td>assign('d','b')</td>
<td>vP('b','main@2')</td>
</tr>
<tr>
<td>load</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load('z','f','q')</td>
<td></td>
<td>vP('c','main@3')</td>
</tr>
</tbody>
</table>
Step 5: Apply rules until convergence

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

Relations
\[ vP_0 \]
\[ vP_0(x',\text{main}@1'). \]
\[ vP_0(y',\text{main}@2'). \]
\[ vP_0(z',\text{main}@3'). \]
\[ \text{store} \]
\[ \text{store}(c',f',d'). \]
\[ \text{load} \]
\[ \text{load}(z',f',q'). \]

Step 5: Apply rules until convergence

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

Relations
\[ vP_0 \]
\[ vP_0(x',\text{main}@1'). \]
\[ vP_0(y',\text{main}@2'). \]
\[ vP_0(z',\text{main}@3'). \]
\[ \text{store} \]
\[ \text{store}(c',f',d'). \]
\[ \text{load} \]
\[ \text{load}(z',f',q'). \]
Virtual Method Invocation

- 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
  - t’s first superclass that defines n

```
void draw (shape s) {
    int i = s.lines();
    ...
}
```

Virtual Method Invocation

- Class hierarchy analysis cha (t, n, m)
  - Simple analysis: can determine the type if the program only allocates one type of objects.
Pointer Analysis
Can Improve Call Graphs

Discover points-to results and methods invoked on the fly

\text{invokes (s, m): statement s calls method m}
\text{hType (h, t): h has type t}
\text{invokes (s, m) :- "s: v.n (...)" & pts (v, h) & hType (h, t) & cha (t, n, m)}

\text{actual (s, i, v): v is the ith actual parameter in call site s.}
\text{formal (m, i, v): v is the ith formal parameter declared in method m.}
\text{pts(v, h) :- invokes (s, m) & formal (m, i, v) & actual (s, i, w) & pts (w, h)}
4. Context-Sensitive Pointer Analysis

Even without recursion, # of contexts is exponential!
Recursion

Top 20 Sourceforge Java Apps

Number of Clones

![Graph showing the relationship between size of program and number of clones.](image)
Cloning-Based Algorithm

- Apply the context-insensitive algorithm to the program to discover the call graph
- Find strongly connected components
- Create a “clone” for every context
- Apply the context-insensitive algorithm to cloned call graph
- Lots of redundancy in result
- Exploit redundancy by clever use of BDDs (binary decision diagrams)

Whaley&Lam, PLDI 2004 (best paper award)

Automatic Analysis Generation

Programmer:
Security analysis in 10 lines

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

PQL

Datalog

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

BDD operations

BDD: 10,000s-lines library

1000s of lines
1 year tuning
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)