Lecture 10

Pointer Analysis

1. Datalog
2. Context-insensitive, flow-insensitive pointer analysis
3. Context sensitivity

Readings: Chapter 12
Pointer Analysis to Improve Security

- Top web application security vulnerabilities
  - SQL injection, cross-site scripting
- User input accessing databases
- Information flow analysis (taint analysis)
- Sound analysis that found errors in 8 out of 9 apps

\[
p_1 = \text{req.getParameter ( )};
\]

\[
\text{stmt.executeQuery (p_2);}
\]

\[p_1 \text{ and } p_2 \text{ point to same object?}
\]

Pointer alias analysis
Automatic Analysis Generation

Programmer:
Security analysis in 10 lines

Compiler writer:
Ptr analysis in 10 lines

PQL

Datalog

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

1000s of lines
1 year tuning

BDD operations

BDD: 10,000s-lines library

Compiler writer:
Ptr analysis in 10 lines

Programmer:
Security analysis in 10 lines

PQL

Datalog

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

1000s of lines
1 year tuning

BDD operations

BDD: 10,000s-lines library
Goals of the Lecture

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

- Power of languages and abstractions

- Elegant abstractions
  - Logic programming
  - BDDs: Binary decision diagrams
    (Most-cited CS paper a few years ago)
1. Why a Deductive Database?

- 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
Datalog Basics

- $p( X_1, X_2, \ldots X_n)$
  - $p$ is a predicate
  - $X_1, X_2, \ldots X_n$ are terms such as variables or constants

- A predicate can be viewed as a relation
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

Example Call graph edges:

A \rightarrow B, A \rightarrow C, A \rightarrow D, B \rightarrow D, C \rightarrow D

Predicate:
calls(A,B)
calls(A,C)
calls(A,D)
calls(B,D)
calls(C,D)
Datalog Programs: Set of Rules (Intensional DB)

- $H \text{ :- } B_1 \& B_2 \ldots \& B_n$
- 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
2. 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();
        q = f();
    }
    A: a = new O ();
    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: formal = actual
  - Return: actual = return value
Pointer Analysis Rules

Object creation
pts(v, h) :- “h: T v = new T()”.

Assignment
pts(v₁, h₁) :- “v₁ = v₂” & pts(v₂, h₁).

Store
hpts(h₁, f, h₂) :- “v₁.f = v₂” & pts(v₁, h₁) & pts(v₂, h₂).

Load
pts(v₂, h₂) :- “v₂ = v₁.f” & pts(v₁, h₁) & hpts(h₁, f, h₂).
Pointer Alias Analysis

- Specified by a few Datalog rules
  - Creation sites
  - Assignments
  - Stores
  - Loads

- Apply rules until they converge
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

```java
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 program only allocate one type of objects.

```java
do draw (shape s) {
    int i = s.lines();
    ...
}
```
Pointer Analysis
Can Improve Call Graphs

Discover points-to results and methods invoked on the fly

invokes (s, m): statement s calls method m
hType (h, t): h has type t

\[
\text{invokes} \ (s, \ m) \ :- \ "s: \ v.n \ (\ldots)" \ & \ \text{pts} \ (v, \ h) \ & \\
\hType \ (h, \ t) \ & \ \text{cha} \ (t, \ n, \ m)
\]

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.

\[
\text{pts} (v, \ h) \ :- \ \text{invokes} \ (s, \ m) \ & \\
\text{formal} \ (m, \ i, \ v) \ & \ \text{actual} \ (s, \ i, \ w) \ & \\
\text{pts} \ (w, \ h)
\]
3. Context-Sensitive Pointer Analysis

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

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

$10^0$  $10^4$  $10^8$  $10^{12}$  $10^{16}$

L10: Pointer Analysis
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: Ptr analysis in 10 lines

PQL

Datalog

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

BDD operations

BDD: 10,000s-lines library

Compiler Writer:
Ptr analysis in 10 lines

Programmer:
Security analysis in 10 lines

1000s of lines
1 year tuning

BDD: 10,000s-lines library