Lecture 12
Pointer Analysis

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

Readings: Chapter 12

A Simple SQL Injection Pattern

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

In Practice

ParameterParser.java:586
String session.ParameterParser.getRawParameter(String name)

```
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)

```
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)

```
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

Inject		Exploit
Parameters		SQL injection
Hidden fields	X	Cross-site scripting
Headers		HTTP splitting
Cookie poisoning		Path traversal

Key: Information Flow

PQL: Program Query Language

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

- Find security errors by monitoring run-time behavior
- Language: Query on the dynamic behavior based on object entities
  - Abstracting away the intermediate information flow operations

Dynamic vs. Static Pattern

Dynamically: o = *req*.getParameter ( );
stmt.executeQuery (o);

Statically: p_1 = *req*.getParameter ( );
stmt.executeQuery (p_2);

p_1 and p_2 point to same object?

Pointer alias analysis

Security Analysis

- 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
- 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:
  - aliases vs. points-to analysis
  - An unbounded number of dynamically allocated objects
- Representation
  - Needs to model each field in an object
  - The state of the information is large
- Precision
  - C, C++ uses address arithmetic.
    - Every pointer can be written to if the address is unknown.
  - A read through an unknown pointer gets all possible values (incl. pointers)
  - A write through an unknown pointer will pollute many pointer variables.
    - Worse if it is not a typed language
  - Imprecision will propagate many points-to relationships
    - The size of the state grows with imprecision
  - Needs whole-program interprocedural analysis
    - Must model parameter passing
    - A callee's side effects depend on the caller's context (applies transitively)

Automatic Conservative Analysis Generation

Programmer:
Security analysis
in 10 lines

PQL

Compiler Writer:
Ptr analysis in 10 lines

Datalog

bddbddb
(**BDD**-**b**ased
deductive **d**ata**b**ase)
with
Active Machine Learning

1000s of lines
1 year tuning

BDD operations

BDD (Binary Decision Diagrams): 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
  - Datalog: A deductive database
    (A database that can make deductions from stored data)
  - BDDs: Binary decision diagrams
    (Most cited CS papers for many years)

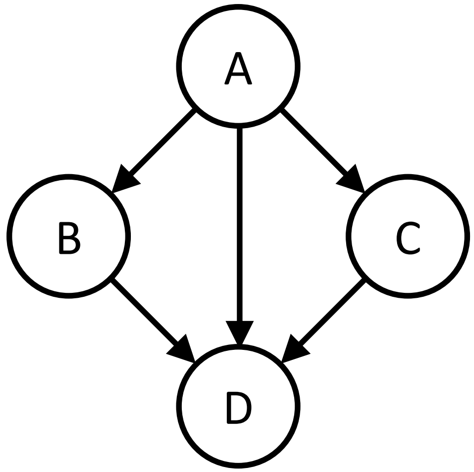
Outline
Pointer Analysis

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: Extensional database
  - Deduced results: Intensional database

Example: Call graph edges Predicate vs. Relation



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

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

Datalog Programs: Set of Rules (Intensional DB)

- H :- B_1 & B_2 … & 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 is not monotonic: 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

```
main () {
    p = f();
    q = f();
}

f () {
    A: a = new O ();
    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

Program Abstraction

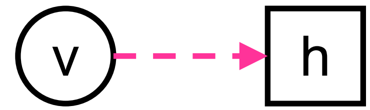
- Allocations h: v = new c
- Store $v_1.f = v_2$
- Loads $v_2 = v_1.f$
- Moves, arguments: $v_1 = v_2$
- Assume: a (conservative) call graph is known a priori for now
  - Call: formal = actual
  - Return: actual = return value

Pointer Analysis Rules

Object creation

pts(v, h) :- "h: T v = new T()".

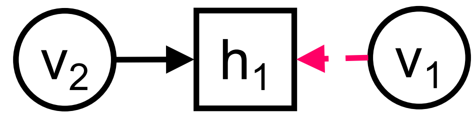
h: T v = new T();



Assignment

$pts(v_1, h_1)$:- "$v_1 = v_2$" & $pts(v_2, h_1)$.

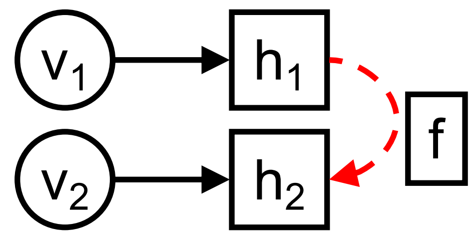
$v_1 = v_2$



Store

$hpts(h_1, f, h_2)$:- "$v_1.f = v_2$" &
$pts(v_1, h_1)$ & $pts(v_2, h_2)$.

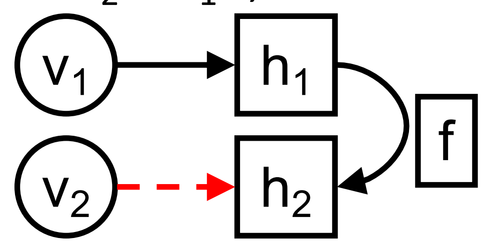
$v_1.f = v_2$;



Load

$pts(v_2, h_2)$:- "$v_2 = v_1.f$" &
$pts(v_1, h_1)$ & $hpts(h_1, f, h_2)$.

$v_2 = v_1.f$;



Pointer Alias Analysis

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

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 (object h_1 field f can point to h_2)

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).

Step 1: Assign numbers to elements in domain

```
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

V

'x' : 0
'y' : 1
'z' : 2
'a' : 3
'b' : 4
'c' : 5
'd' : 6

H

'main@1' : 0
'main@2' : 1
'main@3' : 2

F

'f' : 0

1. Perfect answer: What do x.f and z.f point to?
2. What does this algorithm produce?
   What do x.f and z.f point to?

Step 2: Extract initial relations (EDB) from program

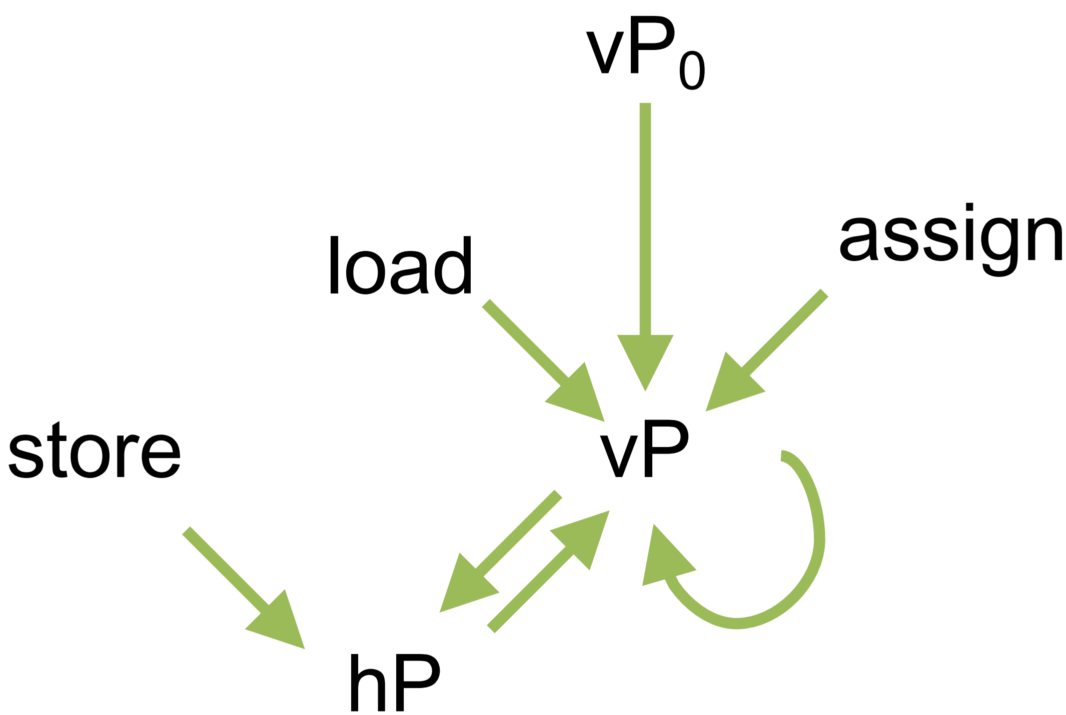
```
void main() {
  x = new C();
  y = new C();
  z = new C();
  m(x,y);
  n(z,x);
  q = z.f;
}

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

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

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').

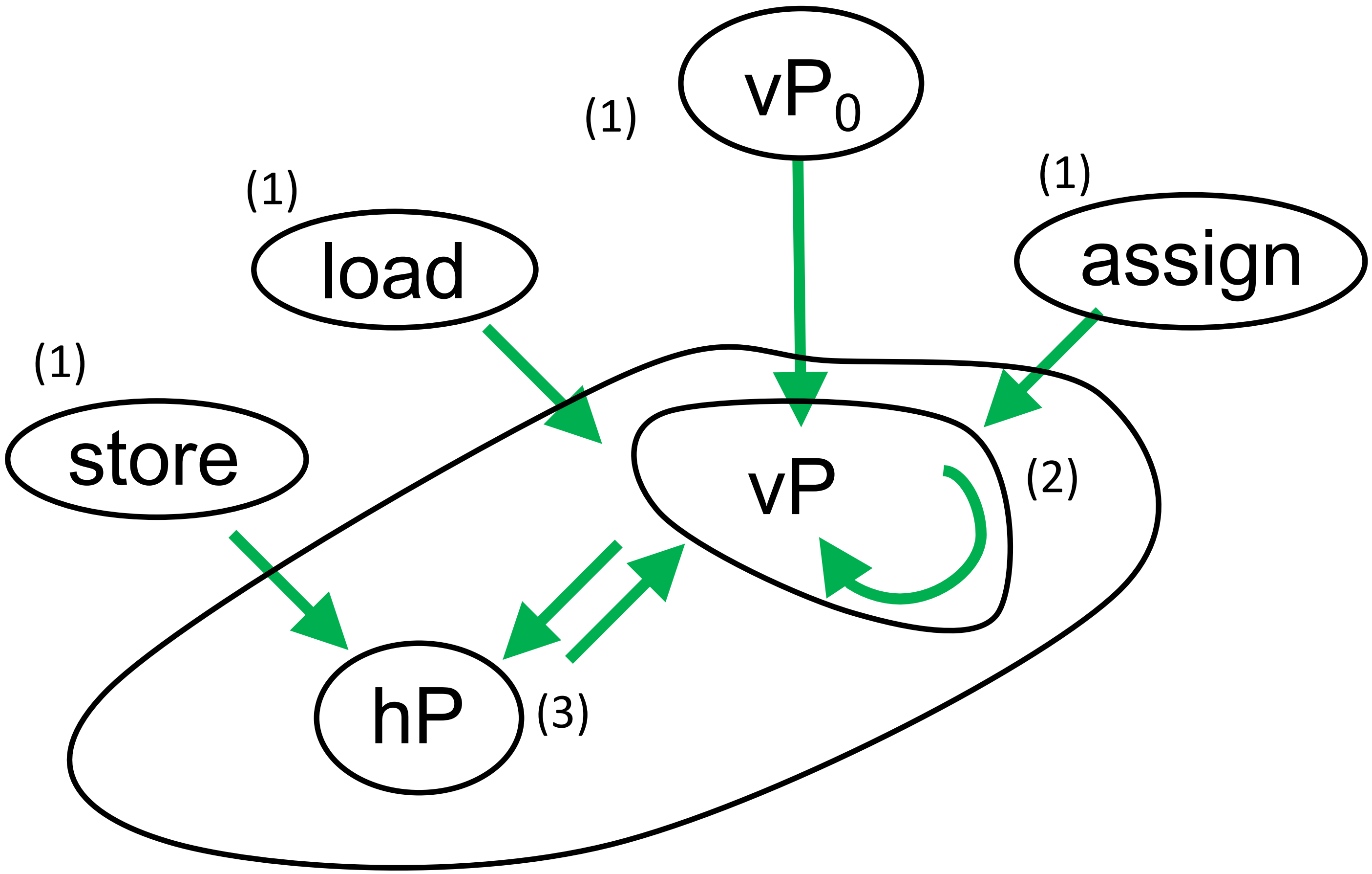
Step 3: Generate Predicate Dependency Graph

Rules

$vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.



Step 4: Determine Iteration Order



In topological sort of strongly connected components
Self-cycles before bigger cycles

Step 5: Apply rules until convergence

Rules

$vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.

```
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;  }
```

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

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

load

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

assign

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

vP

hP

Step 5: Apply rules until convergence

Rules

$vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.

```
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;  }
```

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

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

load

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

assign

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

vP

vP('x','main@1').
vP('y','main@2').
vP('z','main@3').

hP

Step 5: Apply rules until convergence

Rules

$vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.

```
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;  }
```

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

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

load

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

assign

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

vP

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

hP

Step 5: Apply rules until convergence

Rules

$vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.

```
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;  }
```

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

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

load

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

assign

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

vP

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

hP

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).

```
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;  }
```

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

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

load

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

assign

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

vP

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

hP

hP('main@1','f','main@1').
hP('main@1','f','main@2').
hP('main@3','f','main@1').
hP('main@3','f','main@2').

Step 5: Apply rules until convergence

Rules

$vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.

```
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;  }
```

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

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

load

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

assign

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

vP

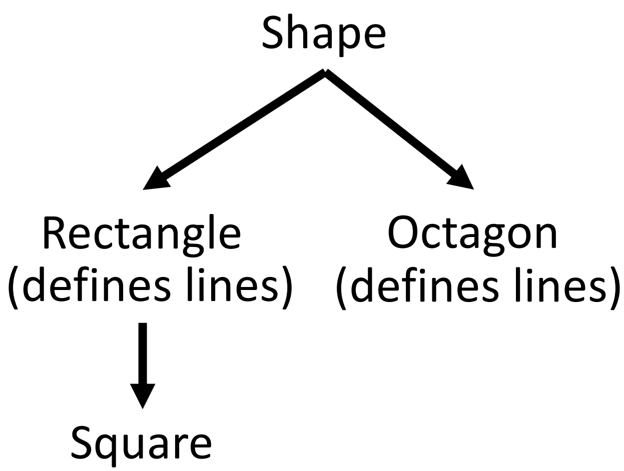
vP('x','main@1').
vP('y','main@2').
vP('z','main@3').
vP('a','main@1').
vP('d,'main@1').
vP('b','main@2').
vP('c','main@3').
vP('c','main@1').
vP('d','main@2').
vP('q','main@1').
vP('q','main@2').

hP

hP('main@1','f','main@1').
hP('main@1','f','main@2').
hP('main@3','f','main@1').
hP('main@3','f','main@2').

Call Graphs

- Previous algorithm assumes an a priori call graph
  - Every possible method is assumed to be invoked
- Virtual method invocations




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

- Define 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

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 SNA 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 (…)" & pts (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.

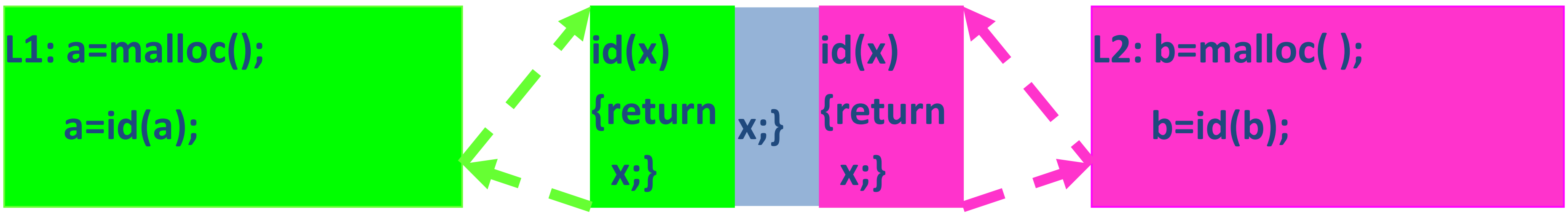
  pts(v, h) :- invokes (s, m) & formal (m, i, v) & actual (s, i, w) & pts (w, h)

Outline

Pointer Analysis

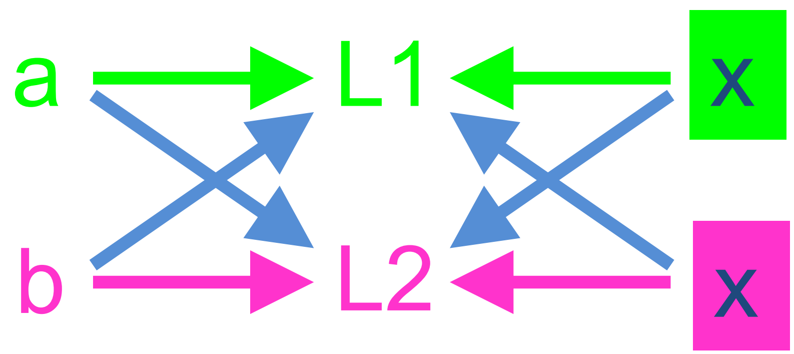
1. Motivation: security analysis
2. Datalog
3. Context-insensitive, flow-insensitive pointer analysis
4. Context sensitivity

4. Context-Sensitive Pointer Analysis

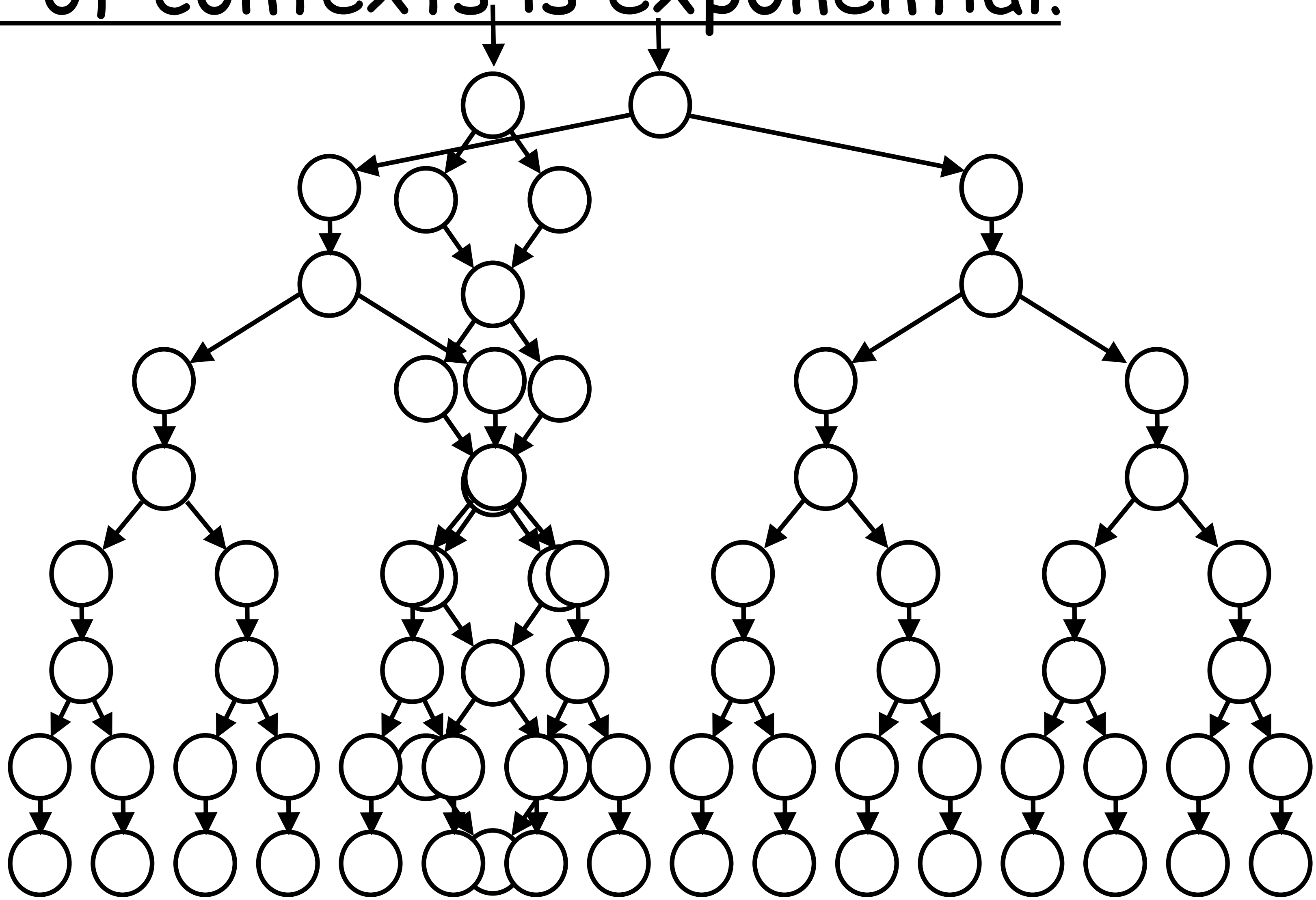


context-sensitive

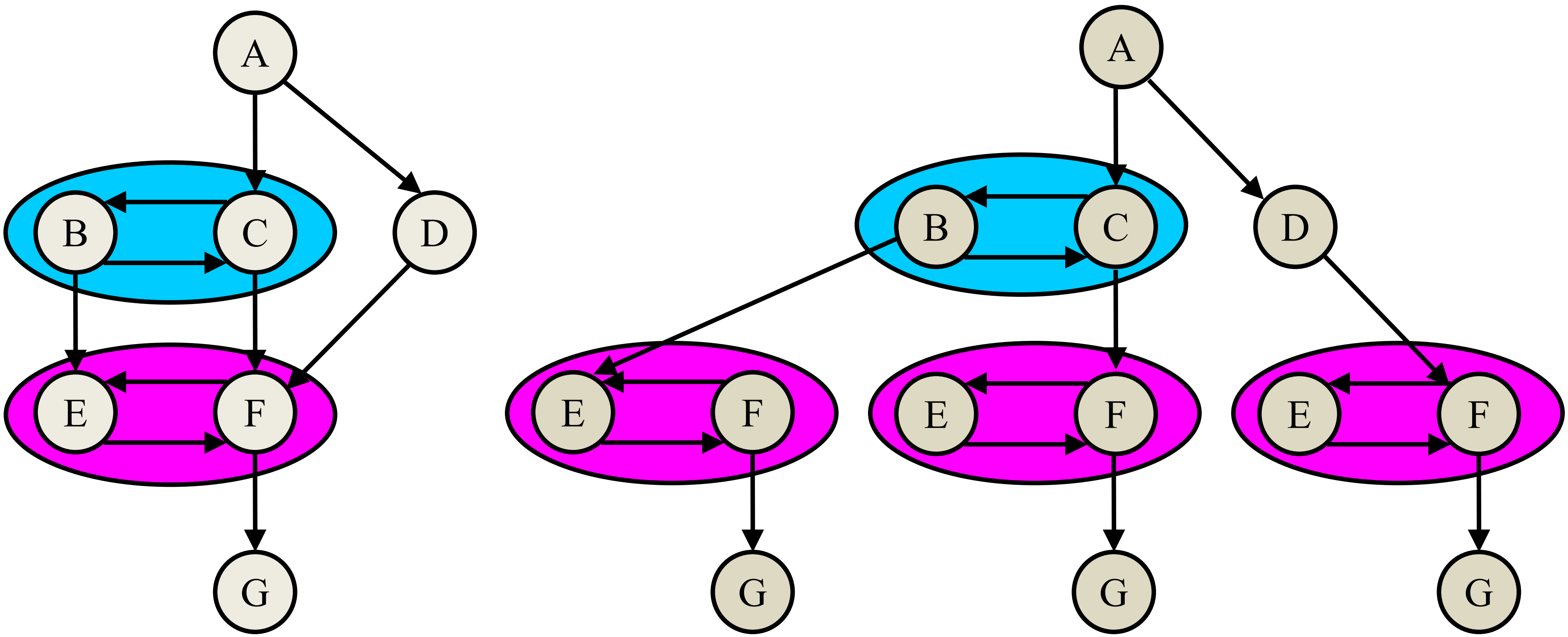
context-insensitive



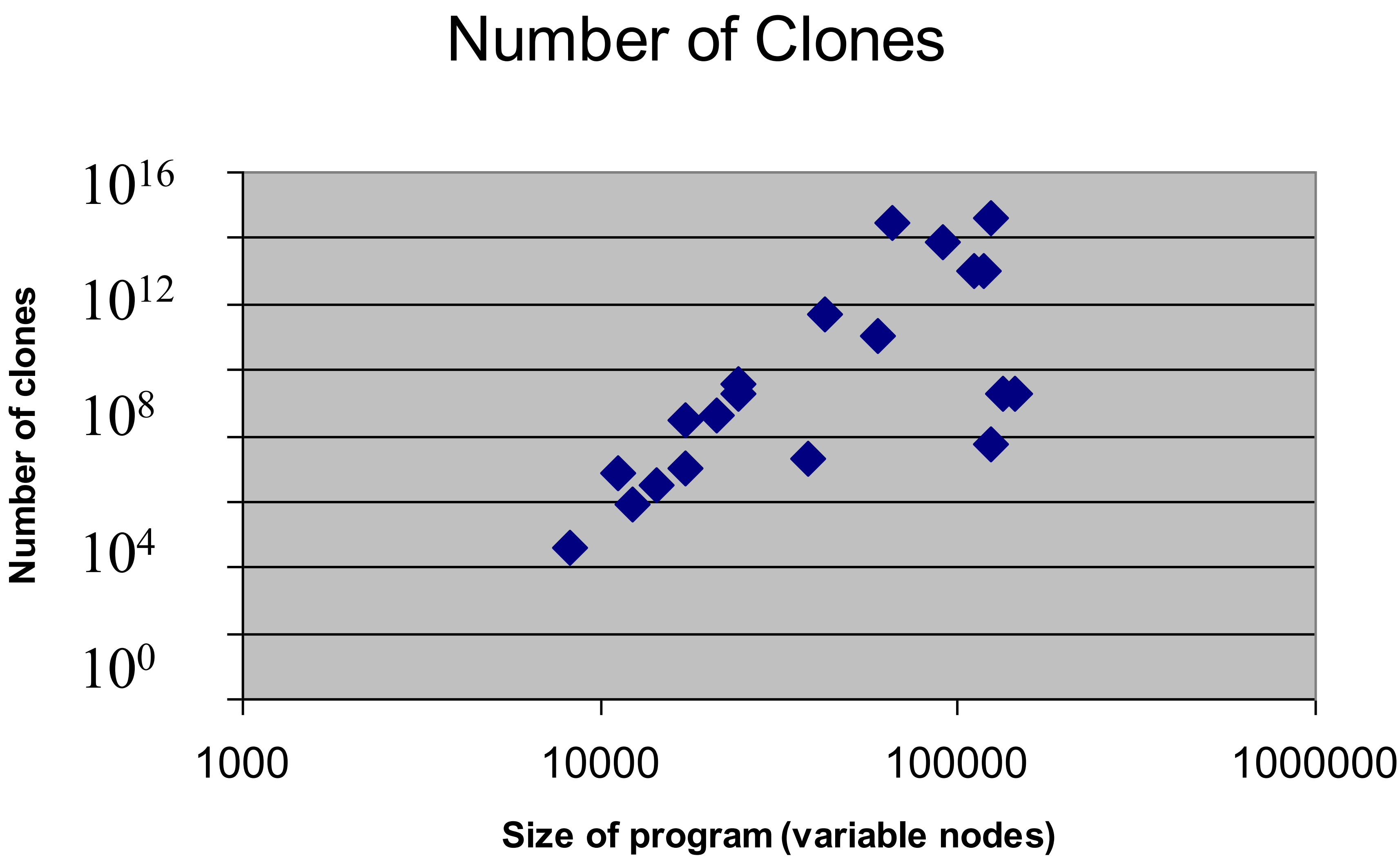
Even without recursion, # of contexts is exponential!



Recursion



Top 20 Sourceforge Java Apps



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)

Whaley&Lam, PLDI 2004 (best paper award)

Automatic Conservative Analysis Generation

Programmer:
Security analysis
in 10 lines

PQL

↓

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

Datalog

↓ **bddbddb**
(**BDD**-**b**ased
deductive **d**ata**b**ase)
with
Active Machine Learning

1000s of lines
1 year tuning

BDD operations

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

General Lessons

- Compilers are about making abstraction choices
  - Must tackle the critical issue: interprocedural analysis
  - Flow sensitivity is not the only way to go
    - Object-oriented programming languages has small function bodies
    - Many local variables are assigned only once!
  - Biggest limitations:
    - Naming of objects: allocation sites are too crude
    - Cannot distinguish between instances of objects used in the same way e.g. a linked list of objects
- The importance of reuse
  - Languages: Datalog
  - Libraries: BDDs (next class)