Lecture 16

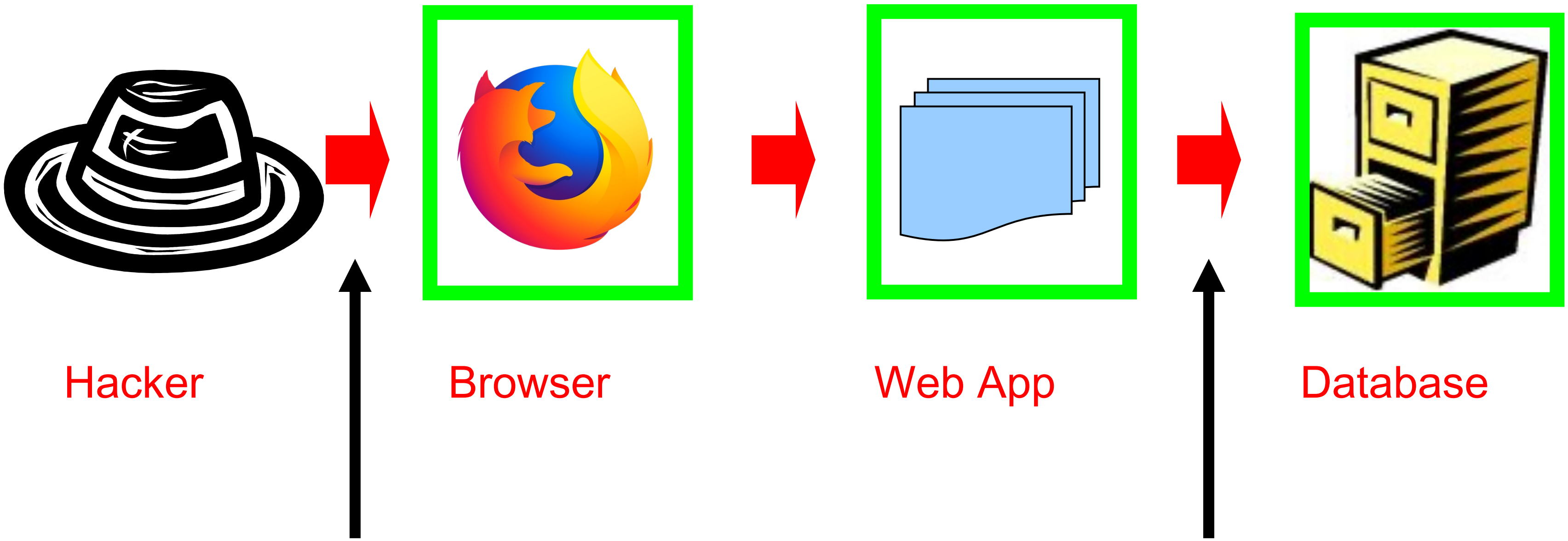
Pointer Analysis

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
In Proc. ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, June 2004.
Best Paper Award.

SQL Injection Errors



Give me Bob's credit card #
Delete all records

SQL Injection Pattern

User supplies text

Dynamically:

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

Text controls the database

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

- 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:

```
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

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

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

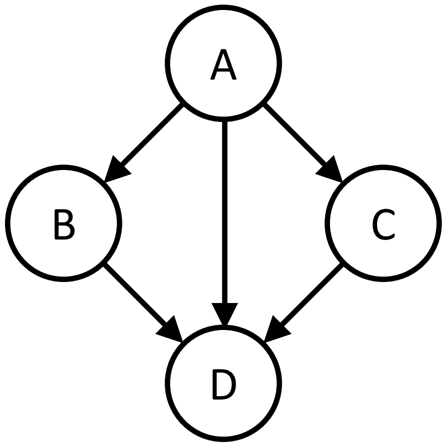
Pointer Analysis

1. 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(X1, X2, … Xn)` — $p(X_1, X_2, \ldots X_n)$
    - $X_1, X_2, \ldots X_n$ 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



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 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

```
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
- Points-to analysis: to compute predicates
  - `vP(v, h)`: variable v can point to object h
  - `hP(h1, f, h2)`: object h_1 field f can point to object h_2

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

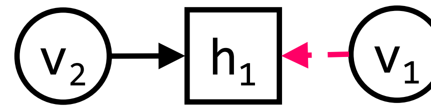
vP(v, h) :- "h: T v = new T()".

h: T v = new T();

Assignment

vP(v_1, h_1) :- "v_1 = v_2", vP(v_2, h_1).

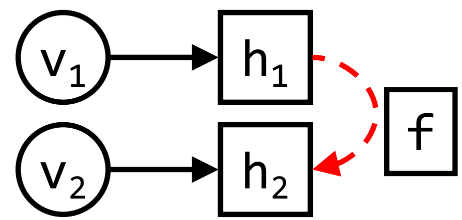
v_1 = v_2;



Store

hP(h_1, f, h_2) :- "v_1.f = v_2",
vP(v_1, h_1), vP(v_2, h_2).

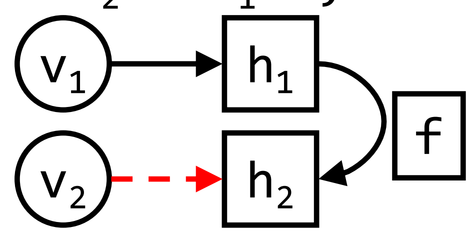
v_1.f = v_2;



Load

vP(v_2, h_2) :- "v_2 = v_1.f",
vP(v_1, h_1), hP(h_1, f, h_2).

v_2 = v_1.f;



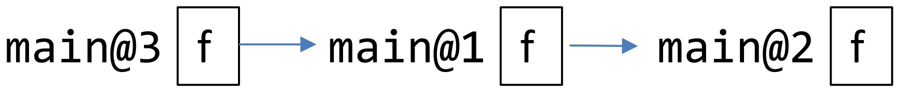
Flow-Insensitive, Context-Insensitive Pointer Alias Analysis

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

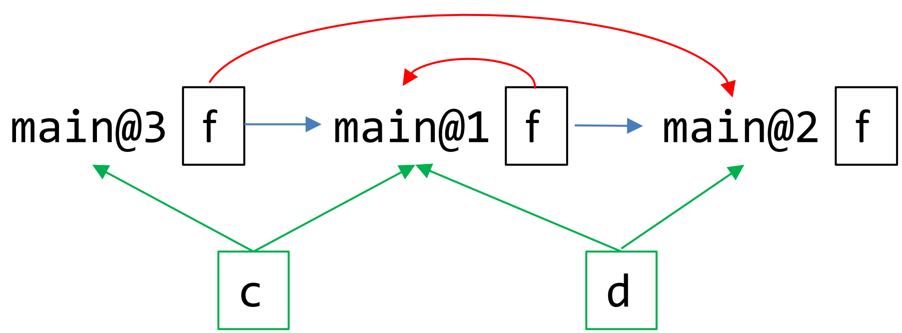
Example

```
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)



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



Pointer Analysis in Datalog

Domains

V = variables

H = heap objects

F = fields

Extensional database EDB (input) relations

```
vP0    (v:V, h:H)          : object allocation sites
assign (v1:V, v2:V)        : assignment instructions (v1 = v2;) and parameter passing
store  (v1:V, f:F, v2:V)   : store instructions (v1.f = v2;)
load   (v1:V, f:F, v2:V)   : load instructions (v2 = v1.f;)
```

Intensional database IDB (computed) relations

```
vP(v:V, h:H)               : variable points-to relation (variable v can point to object h)
hP(h1:H, f:F, h2:H)        : heap points-to relation (object h1 field f can point to h2)
```

Rules

```
vP(v, h)       :- vP0(v, h).
vP(v1, h)      :- assign(v1, v2), vP(v2, h).
hP(h1, f, h2)  :- store(v1, f, v2), vP(v1, h1), vP(v2, h2).
vP(v2, h2)     :- load(v1, f, v2), vP(v1, h1), hP(h1, f, h2).
```

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
 'q' : 7
```

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

```
F
 'f' : 0
```

Step 2: Extract initial relations (EDB) from program

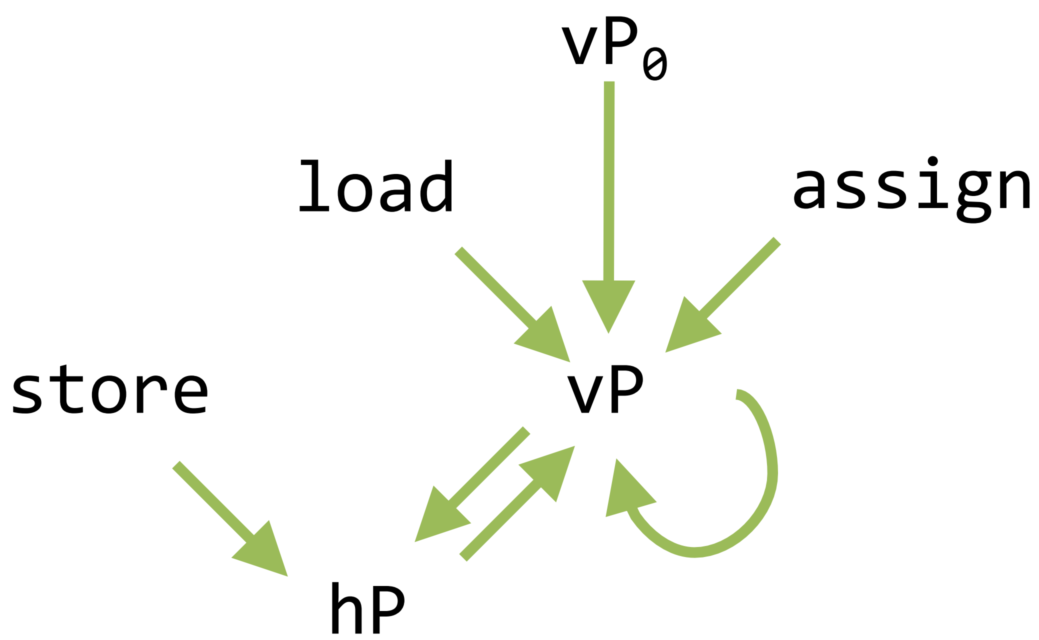
```
void main() {
  x = new C();
  y = new C();
  z = new C();
  m(x,y);
  n(z,x);
  q = z.f;
}

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

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

```
vP0('x', 'main@1').
vP0('y', 'main@2').
vP0('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').
```

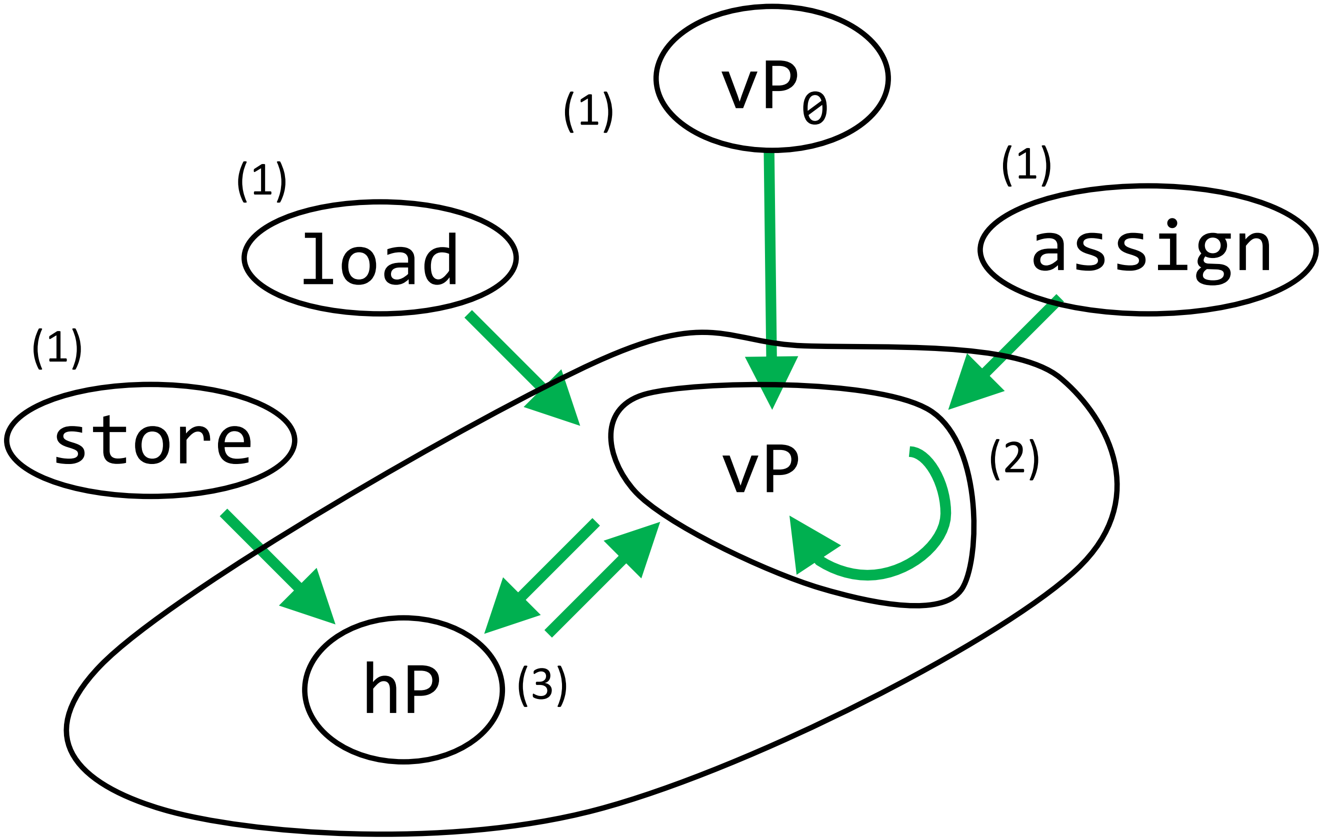
Step 3: Generate Predicate Dependency Graph

Rules

```
vP(v,h) :- vP0(v,h).
vP(v1,h) :- assign(v1,v2), vP(v2,h).
hP(h1,f,h2) :- store(v1,f,v2), vP(v1,h1), vP(v2,h2).
vP(v2,h2) :- load(v1,f,v2), vP(v1,h1), hP(h1,f,h2).
```



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) :- vP0(v,h).
vP(v1,h) :- assign(v1,v2), vP(v2,h).
hP(h1,f,h2) :- store(v1,f,v2), vP(v1,h1), vP(v2,h2).
vP(v2,h2) :- load(v1,f,v2), vP(v1,h1), hP(h1,f,h2).
```

```
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

```
vP0('x','main@1').
vP0('y','main@2').
vP0('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) :- vP0(v,h).
vP(v1,h) :- assign(v1,v2), vP(v2,h).
hP(h1,f,h2) :- store(v1,f,v2), vP(v1,h1), vP(v2,h2).
vP(v2,h2) :- load(v1,f,v2), vP(v1,h1), hP(h1,f,h2).
```

```
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

```
vP0('x','main@1').
vP0('y','main@2').
vP0('z','main@3').
```

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

store

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

load

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

Step 5: Apply rules until convergence

Rules

```
vP(v,h) :- vP0(v,h).
vP(v1,h) :- assign(v1,v2), vP(v2,h).
hP(h1,f,h2) :- store(v1,f,v2), vP(v1,h1), vP(v2,h2).
vP(v2,h2) :- load(v1,f,v2), vP(v1,h1), hP(h1,f,h2).
```

```
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

```
vP0('x','main@1').
vP0('y','main@2').
vP0('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) :- vP0(v,h).
vP(v1,h) :- assign(v1,v2), vP(v2,h).
hP(h1,f,h2) :- store(v1,f,v2), vP(v1,h1), vP(v2,h2).
vP(v2,h2) :- load(v1,f,v2), vP(v1,h1), hP(h1,f,h2).
```

```
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

```
vP0('x','main@1').
vP0('y','main@2').
vP0('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

(red text: imprecision)

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').

(red text: imprecision)

Step 5: Apply rules until convergence

Rules

```
vP(v,h) :- vP0(v,h).
vP(v1,h) :- assign(v1,v2), vP(v2,h).
hP(h1,f,h2) :- store(v1,f,v2), vP(v1,h1), vP(v2,h2).
vP(v2,h2) :- load(v1,f,v2), vP(v1,h1), hP(h1,f,h2).
```

```
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

```
vP0('x','main@1').
vP0('y','main@2').
vP0('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').
```

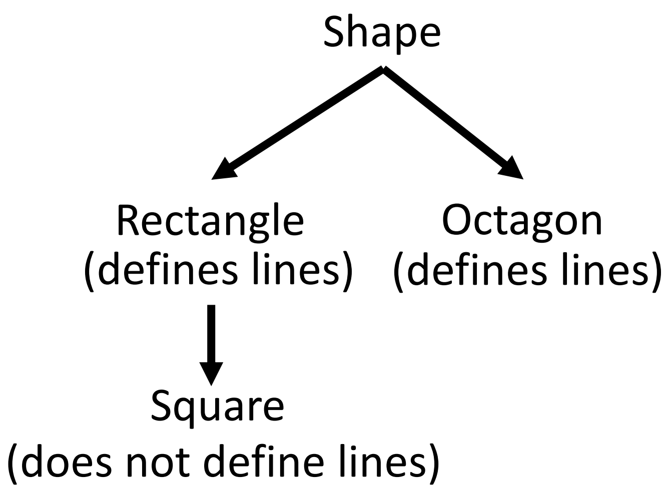
hP

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

(red text: imprecision)

Call Graphs

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




```
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 `i`th actual parameter in call site `s`.
  `formal(m, i, v)` : v is the `i`th formal parameter declared in method `m`.

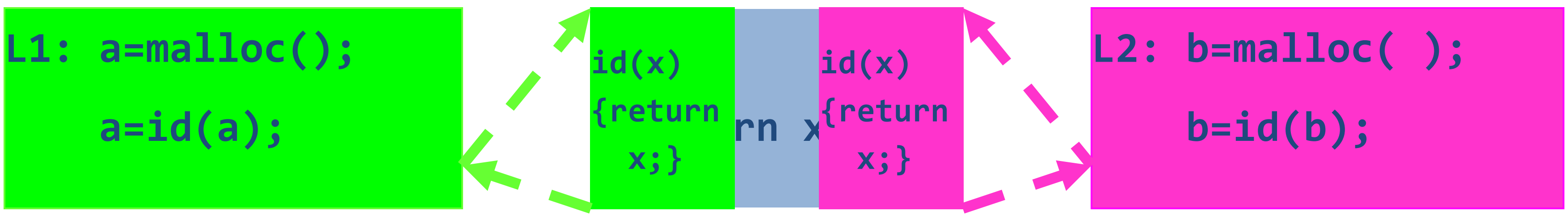
```
vP(v, h):- invokes(s, m), formal(m, i, v), actual(s, i, w), vP(w, h).
```

Outline

Pointer Analysis

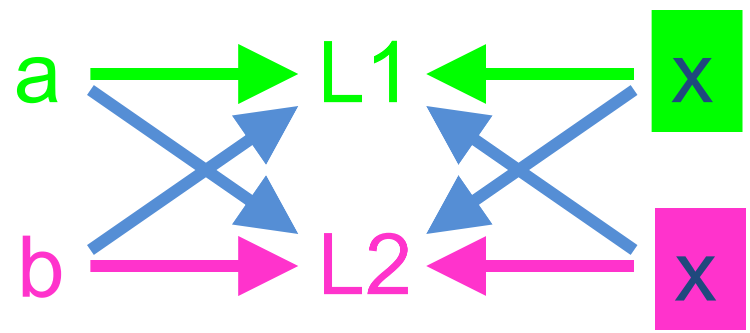
1. Motivation: security analysis
2. Datalog
3. Context-insensitive, flow-insensitive pointer analysis
4. Context sensitivity

4. Context-Sensitive Pointer Analysis

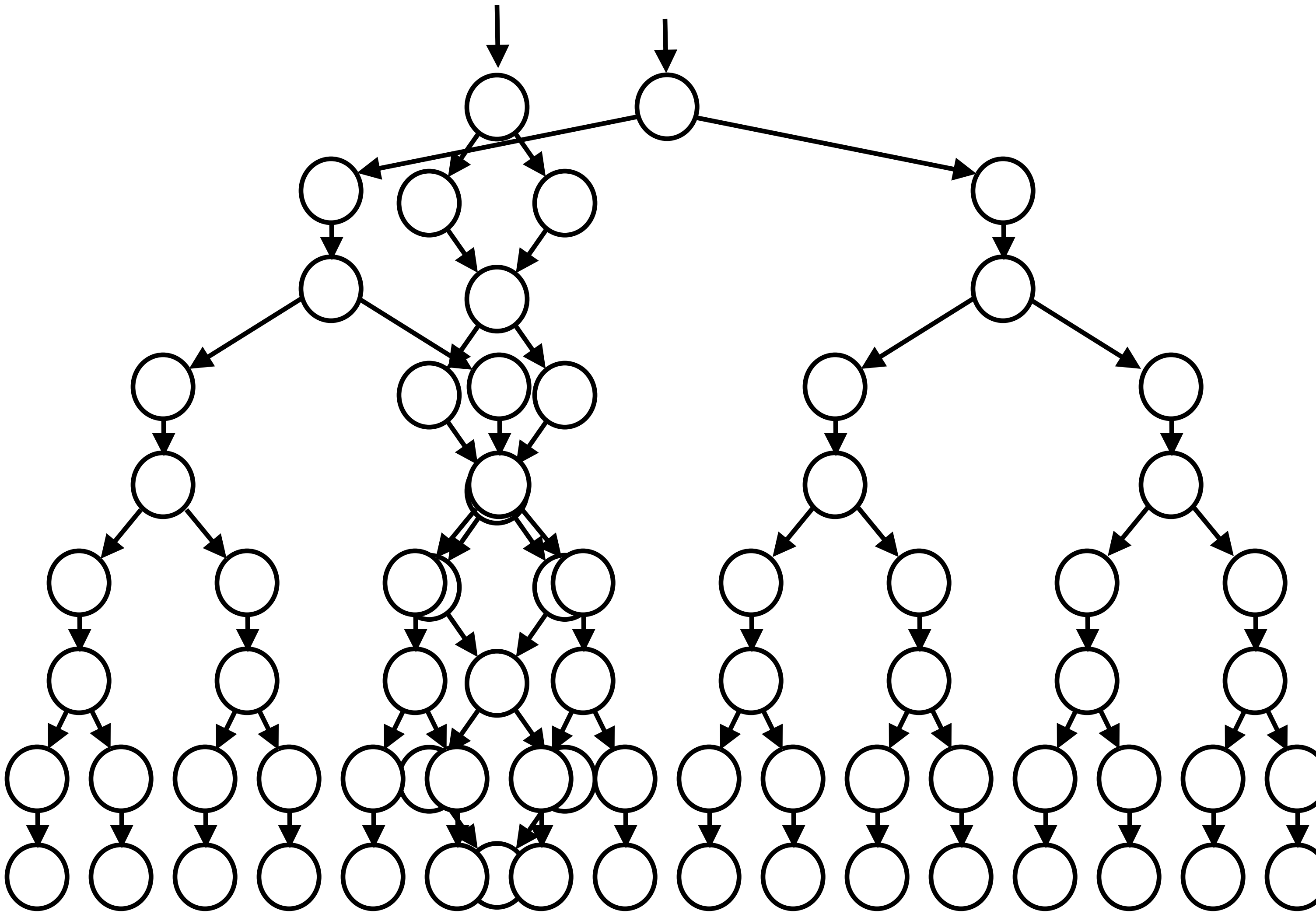


context-sensitive

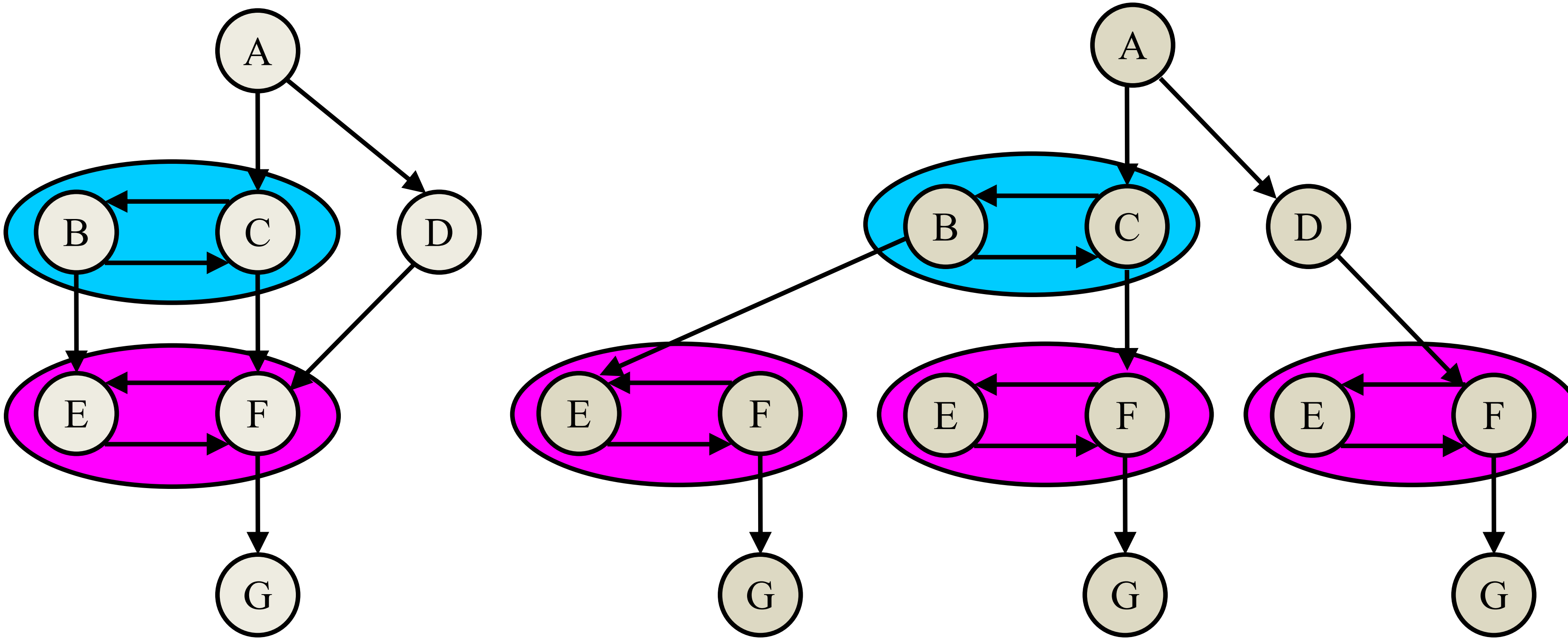
context-insensitive



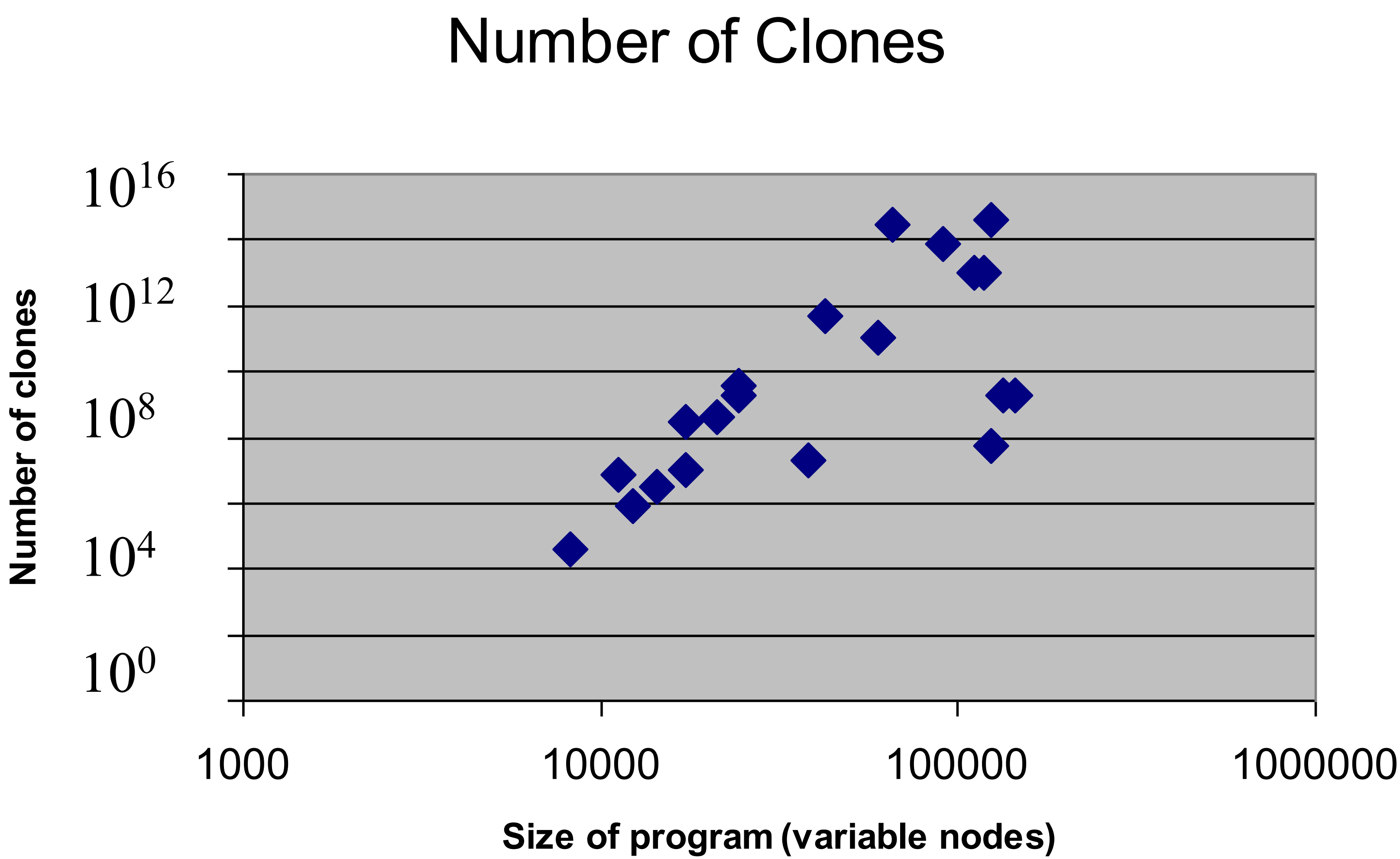
Even without recursion, # of contexts is exponential!



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)

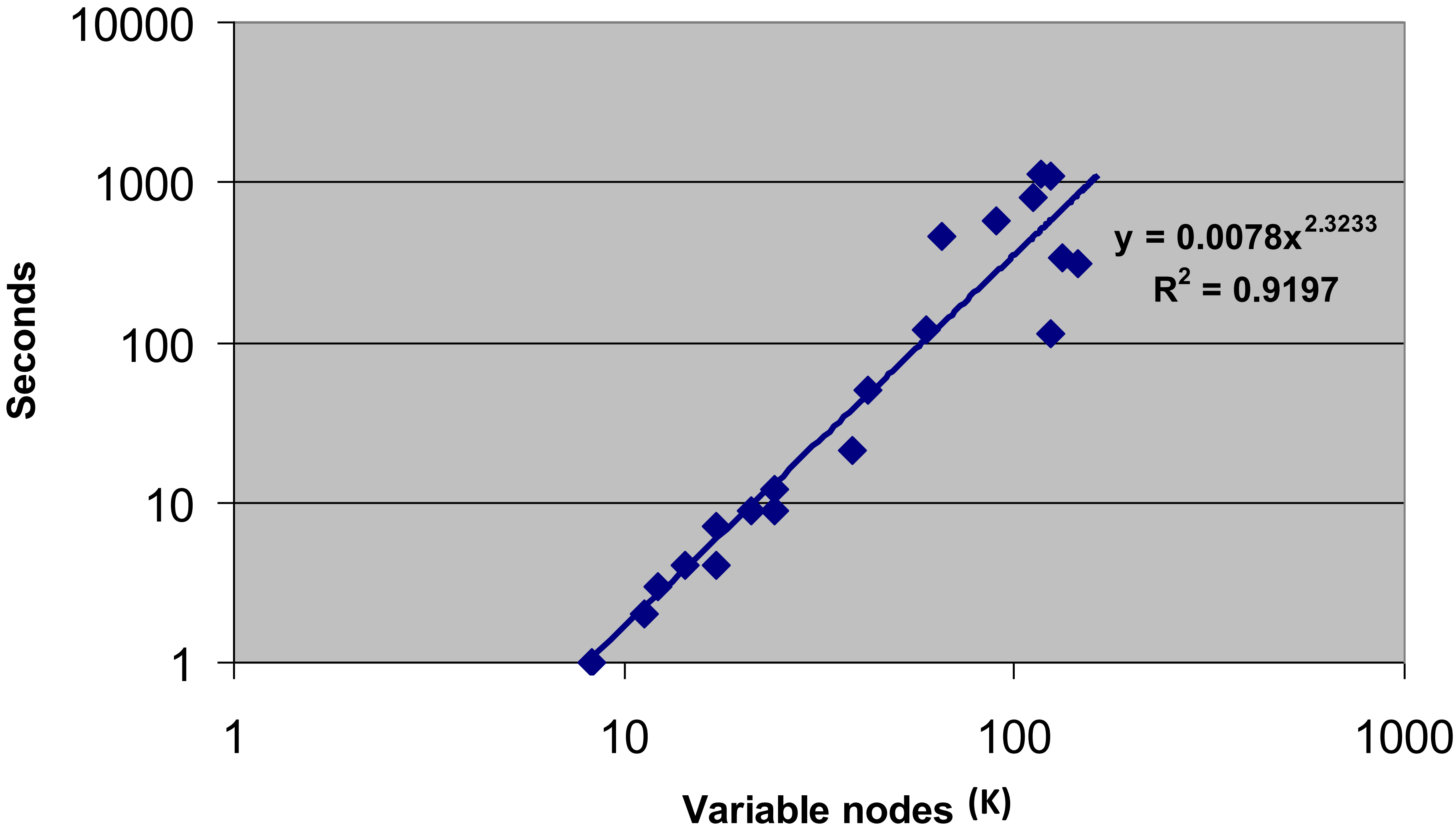
Binary Decision Diagram (BDD)

- 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-**B**ased Deductive DataBase)
  - Datalog rules are implemented with BDD operators

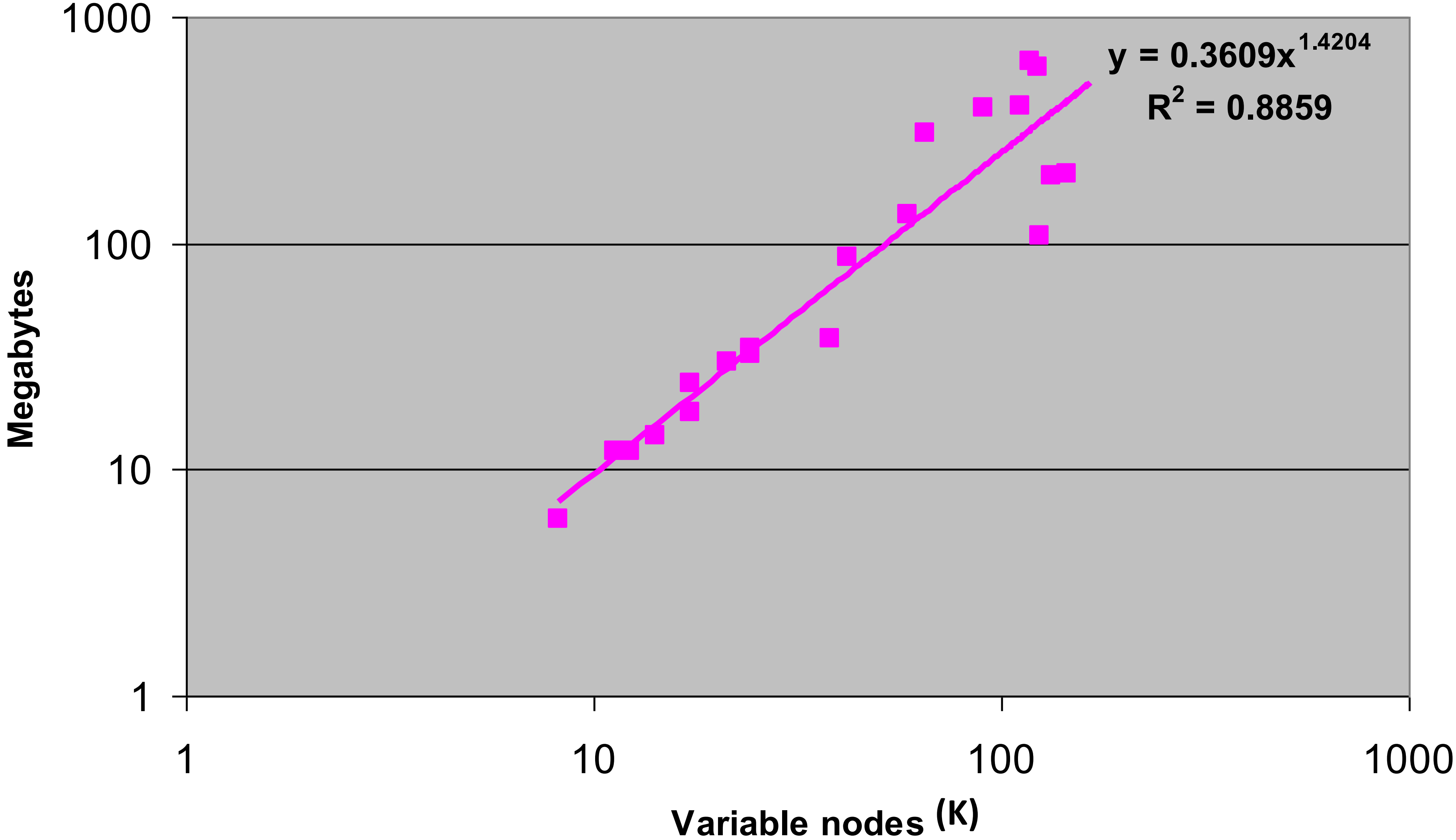
Experimental Results

- 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



Analysis memory



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

	SQL injection	HTTP splitting	Cross-site scripting	Path traversal	Total
Header	0	6	4	0	**10**
Parameter	6	5	0	2	**13**
Cookie	1	0	0	0	**1**
Non-Web	2	0	0	3	**5**
Total	**9**	**11**	**4**	**5**	**29**

Accuracy

Benchmark	Classes	Context insensitive	Context sensitive	False
jboard	264	0	0	0
blueblog	306	1	1	0
webgoat	349	51	6	0
blojsom	428	48	2	0
personalblog	611	460	2	0
snipsnap	653	732	27	12
road2hibernate	867	18	1	0
pebble	889	427	1	0
roller	989	378	1	0
Total	5356	2115	41	12

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 **d**eductive **data** **b**ase) with Active Machine Learning

1000s of lines
1 year tuning

BDD operations

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