Finding Security Vulnerabilities in Java Applications with Static Analysis

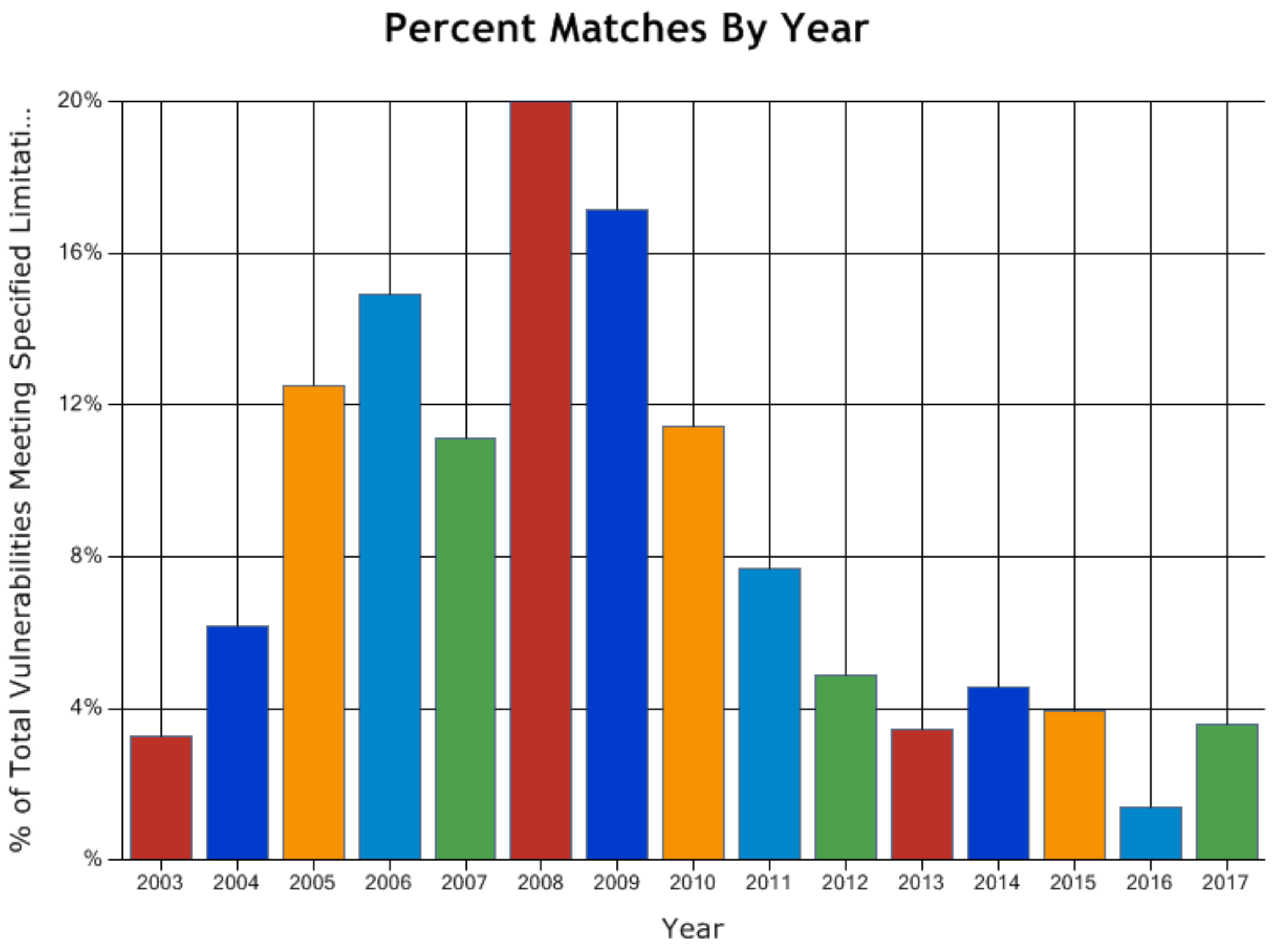
V. Benjamin Livshits and Monica S. Lam

Presented by Chongyang Shi
March 5th, 2018

Background: application vulnerabilities

- Improper application engineering can open up attack surfaces, leading to denial of service, leak of sensitive data, and materialistic loss to users.
- Unchecked inputs controlled by the user is the most prominent attack surface in web applications, covering more than one third of observed attacks (Surf and Shulman, 2001).
- Estimated average cost: USD 4 million for **each** data breach incident (Computer Security Institute, 2002) — with DoS and prevention costs still unaccounted for.
- Common attacks: SQL injection, cross-site scripting, HTTP splitting.

Background: application vulnerabilities



- CVEs associated with SQL injection, as a percentage of all published CVEs between 2003 and 2017.
- Other unchecked input vulnerabilities share similar trends of prominence.

Background: vulnerability analysis

- Three main categories of application vulnerability analysis techniques.
- Penetration Testing (e.g. Arkin et al., 2005)
    - Experienced pen testers with considerable program knowledge perform analysis on the application.
    - Lists of vulnerabilities discovered are likely incomplete.
- Runtime Monitoring (e.g. Dukes et al., 2013)
    - HTTP/HTTPS security proxies and web application firewalls (WAF)
    - Can only prevent attacks with known signs, with privacy implications.
- Static Analysis (used by this research)
    - Automatic path-sensitive analysis on programs.
    - Requires minimal program knowledge.
    - Previous researches could not handle pointers and scalability well.

Research overview

- A Java static security analysis framework capable of defining and detecting a wide range of unchecked input vulnerabilities;
- Context-sensitive static analysis with an improved object-naming scheme;
- Eclipse-integrated vulnerability specification and result viewing interface for programmer usability;
- Evaluations on various popular open source applications and libraries with promising results.

Threat model: injecting malicious data

- Web applications interact with data controlled/defined by the user.
- To attack a web application through unchecked data, a malicious user needs to first inject crafted data into the application through various means:
    - Parameter tempering (HTML forms)
    - URL tempering (encoded GET request parameters)
    - Hidden field tempering (HTML forms)
    - HTTP header manipulation (e.g. referer)
    - Cookie poisoning
    - Command line parameter tempering

Threat model: using injected malicious data

- SQL injections
  - Steal information or deny operation through arbitrary queries.
  - Countered by using prepared statements and parameter binding.
- Cross-site scripting
  - User-defined content improperly placed within dynamically-generated JavaScript on page, enabling phishing and information theft.
  - Countered by properly filtering language features in user inputs.
- HTTP response splitting
- Shell command injection
- Path traversal
  - Countered by input sanitisation and application confinement.

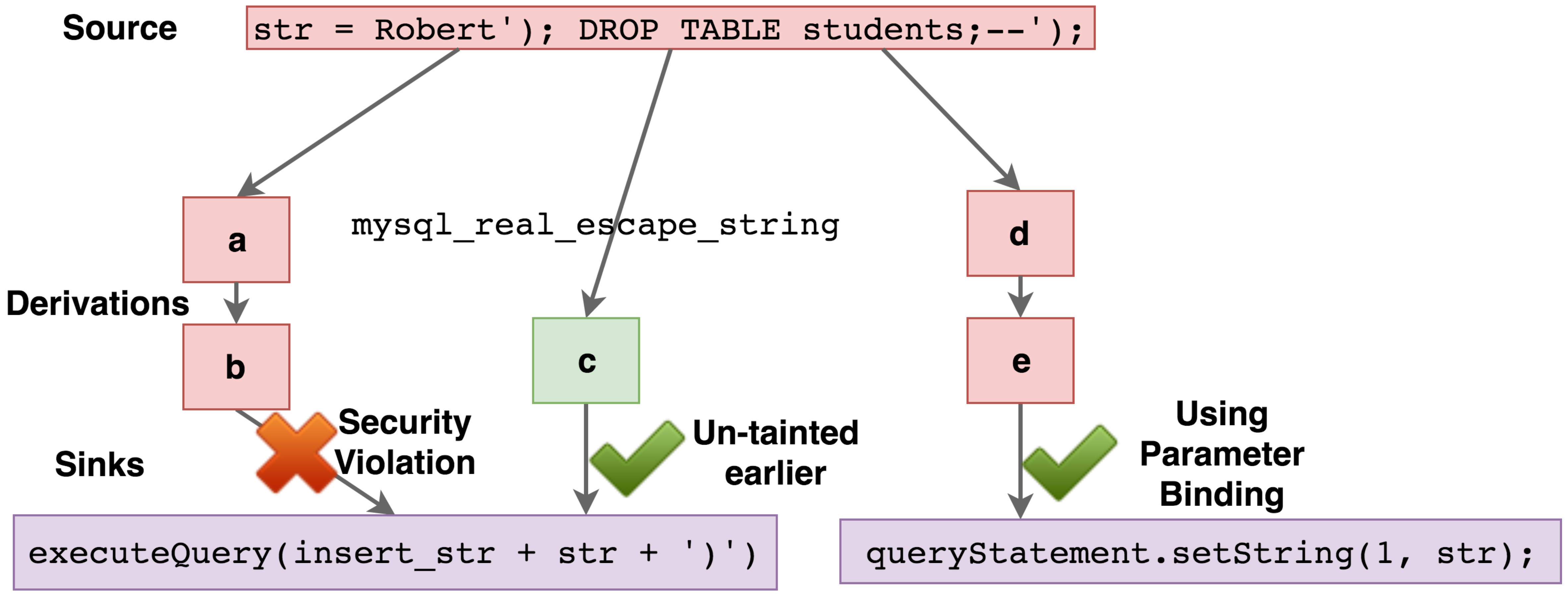
```
HttpServletRequest request = ...;
String userName = request.getParameter("name");
Connection con  = ...
String query    = "SELECT * FROM Users " +
               " WHERE name = ’" + userName + "’";
con.execute(query);
```

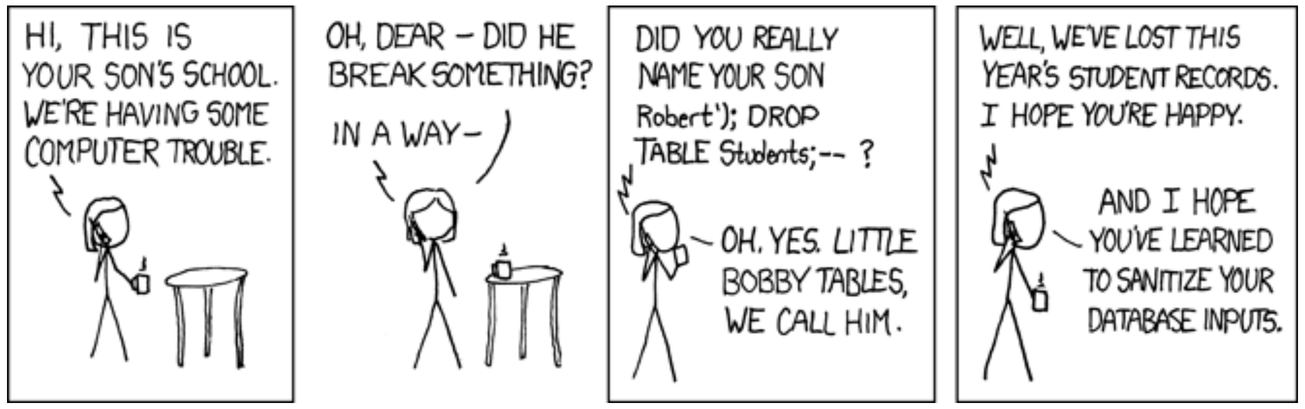
Static analysis: tainted object propagation

- This technique observes tainted objects propagating through *access paths*.
    - Source objects (tainted)
        - Represent user-controlled inputs in the application.
    - Sink objects
        - Represent **unsafe** uses of data in the program;
        - Embedding a string directly in an SQL query creates a sink, but:
        - Binding a string into a prepared statement does *not* create a sink.
    - Derivations
        - Transitive transformations between objects.
        - A source object’s tainted status is preserved over derivations.

Static analysis: security violation

- A **security violation** exists when any sink in the codebase is reachable from a tainted object through derivations.


Source
str = Robert'); DROP TABLE students;--');
a
mysql_real_escape_string
d
Derivations
b
c
e
Security Violation
Un-tainted earlier
Using Parameter Binding
Sinks
executeQuery(insert_str + str + ')')
queryStatement.setString(1, str);


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Static analysis: points-to information

- The user needs to specify sources, sinks, and derivations *descriptors* for the technique to identify vulnerabilities in the code. This requires mapping between descriptors and *points-to* objects in the code.

- The mapping is established through context-sensitive Java points-to-analysis (based on Whaley and Lam, 2004).

- If there exists an **access path** connecting the points-to of a **source** to the points-to of a **sink**, then it is a security violation.

- Points-to information of dynamically generated or loaded objects are approximated from their **allocation sites** (generating or loading code segments).

Static analysis improvement: scalability

- A sound static analysis technique needs to be scalable on codebase variations — it should never miss important vulnerabilities or generate large numbers of false positives.
- Past efforts have been either too conservative in avoiding false positives or unscalable to beyond simple programs.
- This technique: scalable to 10^{14} program contexts in the call graph (approximately 1,000 classes).

Static analysis improvement: precision

- Points-to information already enabled context-sensitivity in this technique (e.g. instances of a class processing tainted and non-tainted data will not be mixed up by the technique).
- Some Java language features require special handling with an improved object naming scheme:
  - Containers (`HashMap`, `HashTable`, `LinkedList` etc.)
    - Can contain tainted objects which require new object names.
  - String manipulations (`String.toLowerCase()` etc.)
    - All objects of the `String` class will be affected if `toLowerCase()` is invoked on a tainted string, requiring new object names for returns;
    - The same problem with synchronising statements on an `Integer`.

Specifying static analysis: PQL

- To allow specification of static analysis in a user-friendly way, the program query language (PQL) was used.
- PQL describes sequences of dynamic events that can lead to tainted objects from a source processed by a sink.
- Results of PQL matching on a codebase can be tracked in Eclipse, allowing easy debugging.

```
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(_)
    | ...
}
```

Evaluation: choice of applications

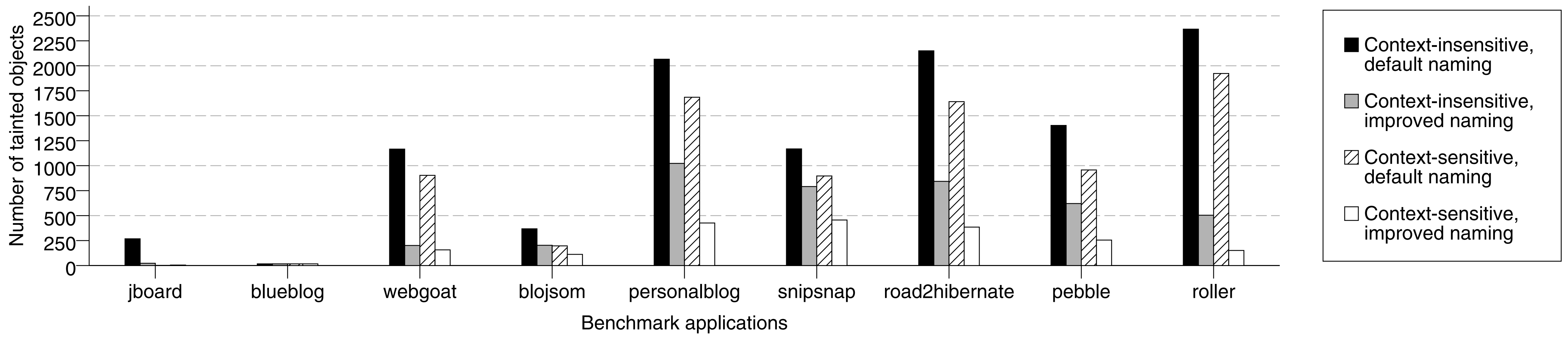
- No established benchmarks at the time for static analysis tools.
- Popular large open source J2EE web applications and libraries were chosen for this technique to be applied on:
  - Web-based bulletin boards and blogging applications;
  - A web application security test case tool (`webgoat`);
  - A popular object persistence library (`road2hibernate`).
- Benchmarking applications with varying code and class sizes.

Benchmark	**Version number**	**File count**	**Line count**	**Analyzed classes**
`jboard`	0.30	90	17,542	264
`blueblog`	1.0	32	4,191	306
`webgoat`	0.9	77	19,440	349
`blojsom`	1.9.6	61	14,448	428
`personalblog`	1.2.6	39	5,591	611
`snipsnap`	1.0-BETA-1	445	36,745	653
`road2hibernate`	2.1.4	2	140	867
`pebble`	1.6-beta1	333	36,544	889
`roller`	0.9.9	276	52,089	989
Total		1,355	186,730	5,356

Evaluation: results

- Precision improvements in context sensitivity and object naming greatly reduced false positives when used in conjunction.
- Improved precision also reduced static analysis run time, rather than increasing it.

	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



Evaluation: quality of results

- All types of unchecked input vulnerabilities were present across codebases.
- Most of the vulnerabilities discovered were easily patchable, and have been patched after responsible disclosure to their authors.
- Serious security vulnerabilities were shown to be present in commonly used libraries, demonstrating the broad coverage of static analysis.
- However, not all vulnerabilities were exploitable in practice, and due to the lack of program knowledge, it is uncertain whether the technique had missed any vulnerabilities.
- The technique ignored control flows that may have been designed to prevent tainted source objects from reaching a sink; but many such control flows in benchmarked codebases did not offer complete protection.

Critique

- The syntax and semantics of PQL are Java-specific, and generalising this technique for other languages will require extensive studies of their standard libraries and popular external libraries to identify sinks and derivations.

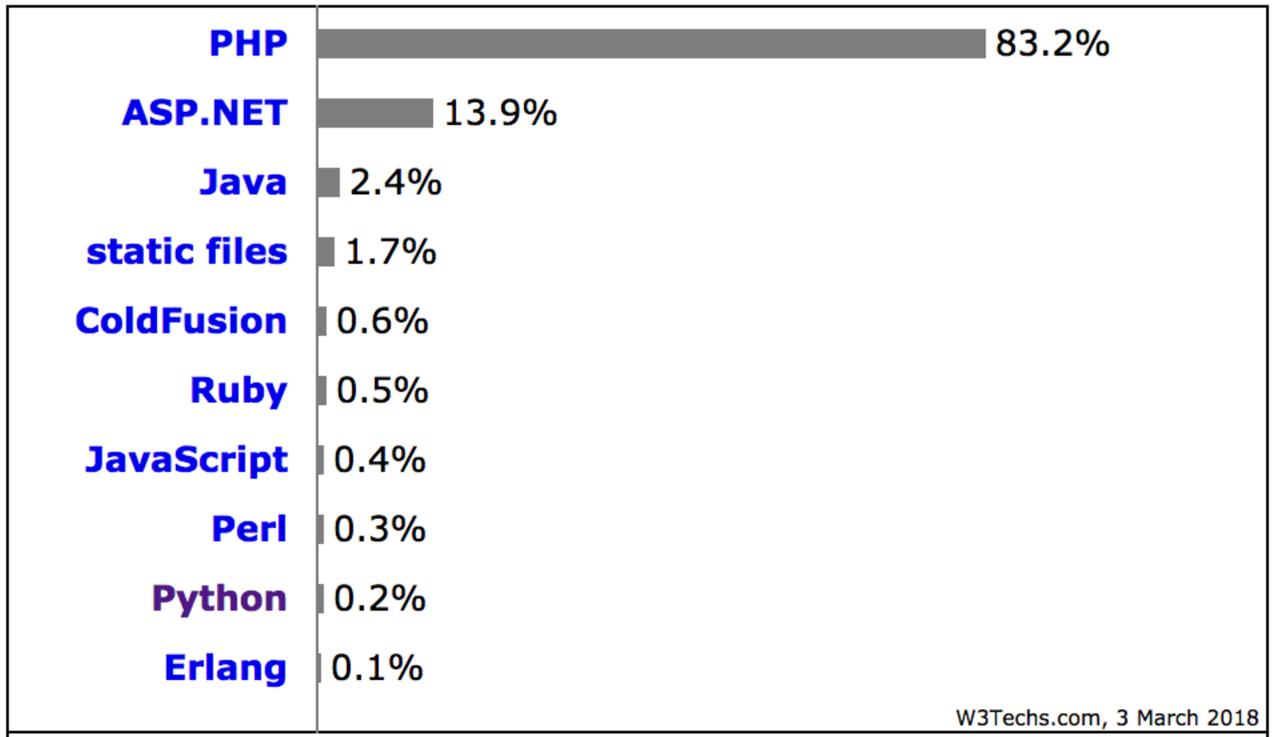
- Adapting this technique for dynamically-typed and weakly-typed languages can be particularly difficult.

- The technique does not modularise the codebase when performing analysis, and wastes considerable computation time on unmodified popular libraries used by many applications.

- The user must be able to specify all possible sinks and derivations through PQL to not miss vulnerabilities, which can be difficult when unfamiliar libraries are used in the user’s codebase.

Current context: Java in web applications

- Java has largely fallen from grace in the web application market.
- Dynamically typed languages such as PHP and JavaScript (Node.js) are now prominent, making static analysis for statically-typed languages less broadly-applicable in the market.
- Static analysis for dynamically-typed languages is under active research (e.g. Jovanovic et al., 2006).



Percentages of websites using various server-side programming languages
Note: a website may use more than one server-side programming language

Current context: static analysis for Android

- However, privacy and malware protection on Android Java applications have taken over as the primary consumers of Java static analysis.

- Awareness of semantics in static analysis is becoming increasingly important.

```
procedure FINDCATCHES(meth, stmt, ex, visiting, stack, cg)
    if (stmt, ex) $\in$ visiting or stmt $\in$ overt then return end if
    visiting $\leftarrow$ visiting $\cup$ (stmt, ex)
    catchBlockStart $\leftarrow$ FINDCOMPATCATCH(meth, stmt, ex)
    if catchBlockStart = null then
        if ISEVENTHANDLER(meth) then
            overt $\leftarrow$ overt $\cup$ stmt
            return
        end if
        for (predStmt, predMeth) $\in$ GETPREDS(cg, meth) do
            if stack $\neq \emptyset$ and (predStmt, predMeth) $\neq$ PEEK(stack) then
                continue
            end if
            newStack $\leftarrow$ stack
            POP(newStack)
            FINDCATCHES(predMeth, predStmt, ex,
                                visiting, newStack, cg)
            if predStmt $\in$ overt then
                overt $\leftarrow$ overt $\cup$ stmt
                return
            end if
        end for
    else
        catchStmts $\leftarrow$ GETCATCHSTMTS(catchBlockStart, meth)
        ANALYZEHANDLER(meth, stmt, catchStmts,
                                visiting, $\emptyset$, stack, cg)
    end if
end procedure
```

- Schmidt, A.D., Bye, R., Schmidt, H.G., Clausen, J., Kiraz, O., Yuksel, K.A., Camtepe, S.A. and Albayrak, S., 2009, June. Static analysis of executables for collaborative malware detection on android. In *Communications, 2009. ICC'09. IEEE International Conference on* (pp. 1-5). IEEE.

- Rubin, J., Gordon, M.I., Nguyen, N. and Rinard, M., 2015, November. Covert communication in mobile applications (t). In Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on (pp. 647-657). IEEE.

- Cito, J., Rubin, J., Stanley-Marbell, P. and Rinard, M., 2016, September. Battery-aware transformations in mobile applications. In Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on (pp. 702-707). IEEE.

Suggested Discussions

- Is it possible to effectively combine static analysis and runtime monitoring/analysis to develop a generalised approach for finding security vulnerabilities in applications of many different languages?
- Is it ultimately possible to establish a good baseline codebase for benchmarking static and dynamic vulnerability analysis techniques?

References

Computer Security Institute. Computer crime and security survey. http://www.gocsi.com/press/20020407.jhtml?requestid=195148, 2002.

Surf, M. and Shulman, A., 2004. How safe is it out there? http://www.imperva.com/download.asp?id=23.

Arkin, B., Stender, S. and McGraw, G., 2005. Software penetration testing. IEEE Security & Privacy, 3(1), pp.84-87.

Dukes, L., Yuan, X. and Akowuah, F., 2013, April. A case study on web application security testing with tools and manual testing. In Southeastcon, 2013 Proceedings of IEEE (pp. 1-6). IEEE.

J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation, pages 131–144, June 2004.

Jovanovic, N., Kruegel, C. and Kirda, E., 2006, May. Pixy: A static analysis tool for detecting web application vulnerabilities. In Security and Privacy, 2006 IEEE Symposium on (pp. 6-pp). IEEE.