Securing Web Applications

Static and Dynamic Information Flow Tracking

Jason Ganzhorn
4/12/2010

The Issues

- SQL Injection
- Cross-site Scripting (XSS)
- Other problems, such as HTTP response splitting…
- Common Theme
  - Unchecked (or “tainted”) data from user reaches security-sensitive operations

Quick Vulnerability Review

What is Cross-Site Scripting?

- Cross-site scripting, in a nutshell, refers to a malicious practice in which code from an evil server is injected into a legitimate page and then run by hapless victims who view the page.
- There are complex variants on this theme, but we won’t be discussing that, since this isn’t a presentation on XSS.

HTTP Response Splitting

- The basic idea behind HTTP response splitting is that the attacker attempts to divide an HTTP header such that the target interprets the response as two.
- Involves an insertion of user-supplied data into the HTTP header.
- If this insertion is successful, the attacker completely controls the content of the second header, which can be used to perform a cache poisoning attack (among other attacks).

SQL Injection

- Basic idea: input from user is incorporated into query on database.
- If the input is unchecked, a malicious attacker can perform nefarious acts such as examining the database structure, reading or writing to database tables, or completely discarding tables.

SQL Injection in the wild

From a recent (Jan. 14, 2010) security advisory for a product called Zenoss Core (http://www.zenoss.com):

getJSONEventsInfo contains multiple SQL Injection vulnerabilities due to improperly sanitized user provided input. The following URL parameters are injectable: severity, state, filter, offset, and count.

A proof of concept request might look like this:

```
/zport/dmd/Events/getJSONEventsInfo?severity
=1&state=1&filter=& offset=0&count=60 into
outfile "/tmp/z‟
```

Obtained from:
http://www.ngenuity.org/wordpress/2010/01/14/ngenuity-2010-001-zenoss-getjsoneventsinfo-sql-injection/

SQL Injection

Tainted Input

- The root of many of these problems is that user-specified (or “tainted”) data is passed into critical code without being checked.
- Why isn’t it being checked?
- Sometimes the source of the data and the “sink” (e.g. a query construction) are widely separated and the connection is not obvious.

Tainted Input (cont'd)

- Consider this example:
  You are programming a networked application, and you decide to use the AwesomeNetwork™ library to handle your database connections.

  However, you are short on time and the documentation for the library is lacking, so you assume that it provides SQL sanitization.

  Voila, you have introduced a SQL injection vulnerability.

How Serious Is It, Really?

- According to the WhiteHat Website Security Statistics Report released in fall 2009, approximately 64% of the 1,364 scanned websites have at least one serious vulnerability.
- There is an average of 6.5 severe unresolved vulnerabilities per website.
- http://www.whitehatsec.com/home/resource/stats.html
- Further interesting reading: http://www.symantec.com/business/theme.jsp?themeid=threatreport

Is There a Solution?

- One solution is to adopt the mindset that user data cannot be trusted and should be sanitized whenever you are passing it to a sink.
- You are still probably going to miss some subtle cases due to lengthy code paths.
- Wouldn’t it be nice if there was an automated way to check for common vulnerability patterns?

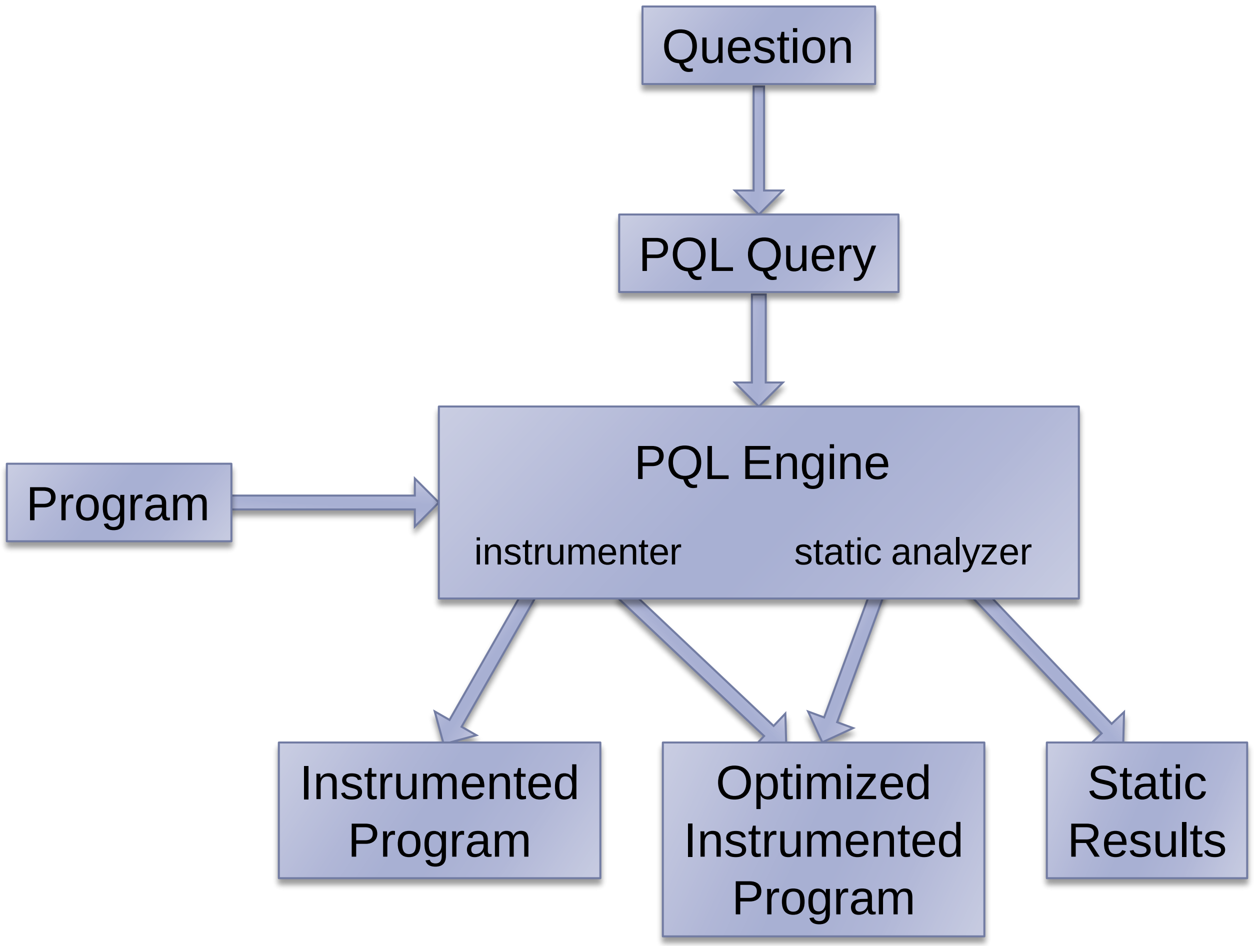
Program Query Language

Securing Web Applications with Static and Dynamic Information Flow Tracking

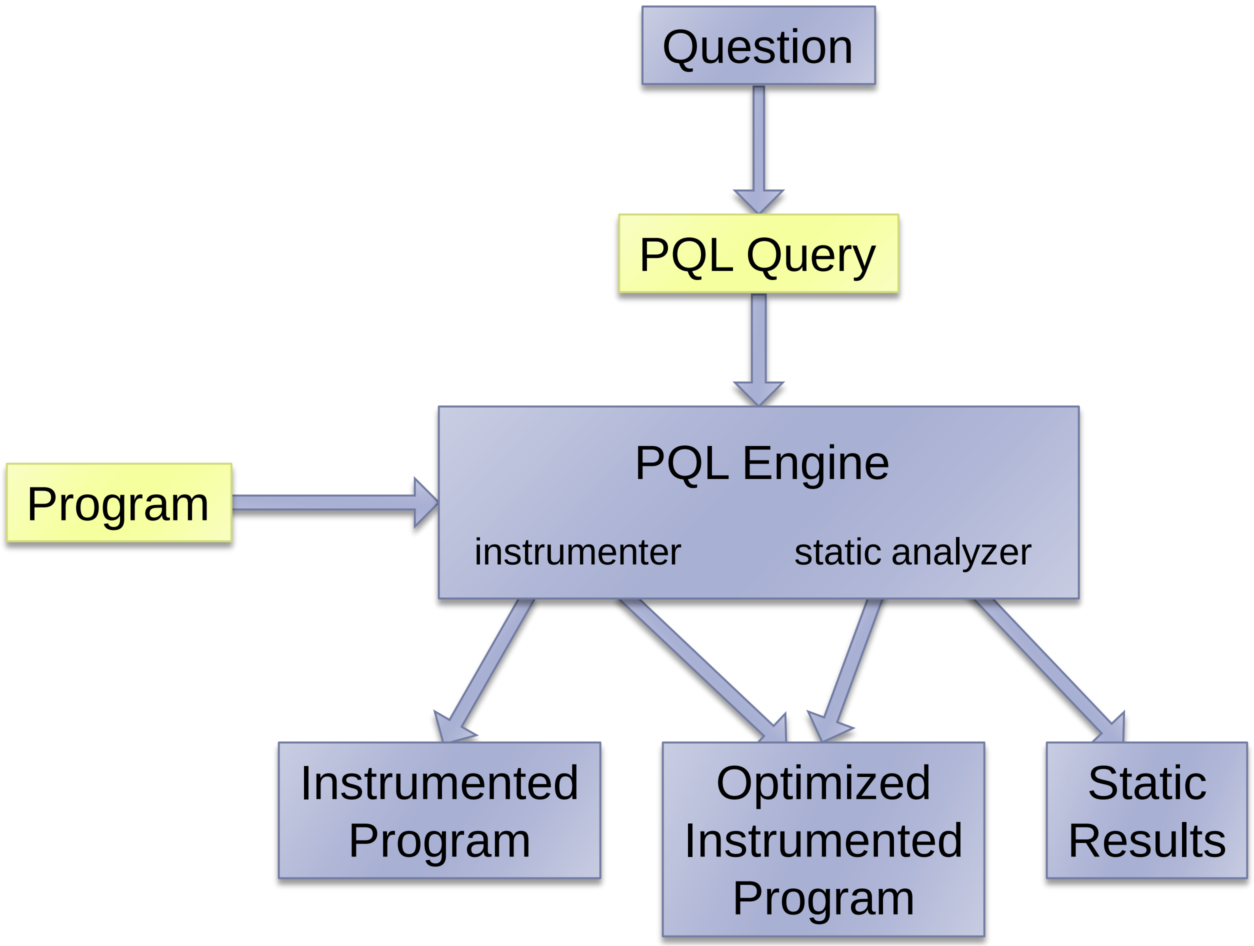
One Possible Solution

- PQL – Program Query Language
- Basic Idea
  - Represent a class of information flow as a pattern, or specification.
  - Use static and dynamic analysis to identify matches against a specification.
  - Allow program to recover from errors gracefully and defend against security breaches.
  - Make this as simple as possible for the programmer.

PQL System Architecture



Programs and PQL Queries



Execution Trace Example

- Here is an extremely simple example of an SQL injection:

```
HttpServletRequest req = /* ... */;
java.sql.Connection conn = /* ... */;
String q = req.getParameter("QUERY");
conn.execute(q);
```

- The corresponding abstract execution trace:
  - CALL o_1.getParameter(o_2)
  - RET o_2
  - CALL o_3.execute(o_2)
  - RET o_4

Another Execution Trace

- Another simple SQL injection example:

```
String read() {
    HttpServletRequest req = /* ... */;
    return req.getParameter("QUERY");
}
/* ... */
java.sql.Connection conn = /* ... */;
conn.execute(read());
```

- The corresponding abstract execution trace:

CALL read()

CALL o_1.getParameter(o_2)

RET o_3

RET o_3

CALL o_4.execute(o_3)

RET o_5

Trace Similarities

- Note how the traces are similar:

	1 CALL read()
1 CALL o_1.getParameter(o_2)	2 CALL o_1.getParameter(o_2)
2 RET o_3	3 RET o_3
	4 RET o_3
3 CALL o_4.execute(o_3)	5 CALL o_4.execute(o_3)
4 RET o_5	6 RET o_5

- PQL can be used to construct queries to look for patterns like these.

Constructing the Specification

- The pattern, specified in PQL:

```
query main()
uses String x;
matches {
     x = HttpServletRequest.getParameter(_);
     Connection.execute(x);
}
```

- The specification is simple, and works for non-adjacent lines of code
- Not complete, however, as it does not match against code that executes a string derived from a parameter.

Things Get More Complicated

- Need to handle derived strings as in this case:

```
HttpServletRequest req = /* ... */;
n = req.getParameter(“NAME”);
p = req.getParameter(“PASSWORD”);
conn.execute(
   “SELECT * FROM logins WHERE name=” +
   n +
   “ AND passwd=” +
   p
);
```

- The PQL specification will have to be more complicated.

New PQL Query, Part 1

- The sub-query to check whether one string is derived from another:

```
query derived (Object x)
uses Object temp;
returns Object d;
matches {
     { temp.append(x); d := derived(temp); }
     | { temp = x.toString(); d := derived(temp); }
     | { d := x; }
}
```

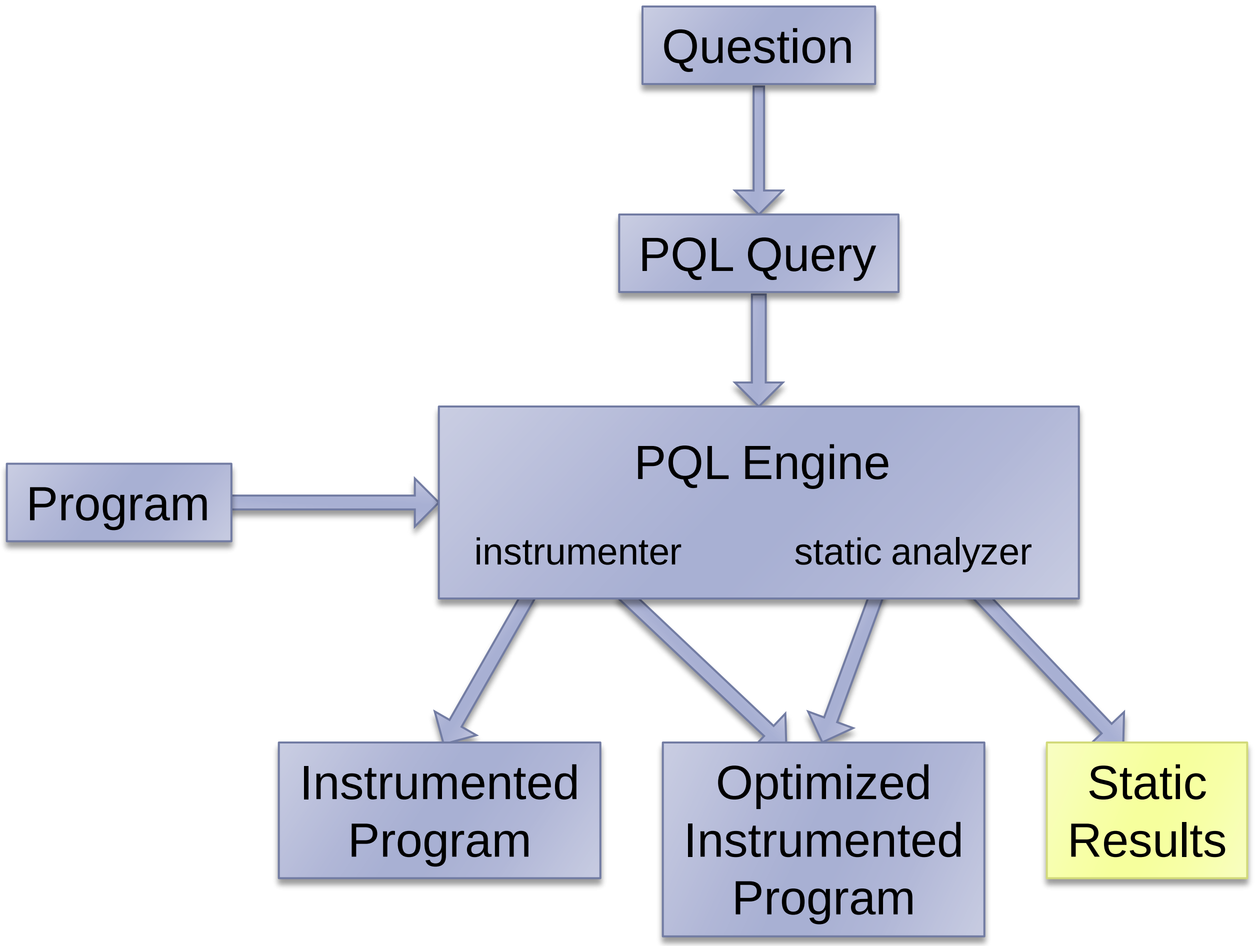
New PQL Query, Part 2

- The new main query:

```
query main()
uses String x, final;
matches {
     x = HttpServletRequest.getParameter(_);
     | x = HttpServletRequest.getHeader(_);
     final := derived(x);
     Connection.execute(final);
}
```

- The new query is a bit more complex, but it does match quite a few more instances of an SQL injection.

Where Are We Right Now?



Benefits So Far

- Difficult for programmers to do analysis (either they don’t like it or they don’t know it).
- However, they are familiar with the program code.
- Analysis specialists are good with analysis, but they don’t know the specific program code.
- Analysts can use PQL to generate generic specifications to catch security flaws.

Building More Robust Applications

- So far we’ve seen PQL queries used to detect vulnerabilities in essentially a static fashion.
- PQL can be used to integrate a specialized query matcher into the executable.
- A match can trigger special code designed to sanitize possibly unsafe input.

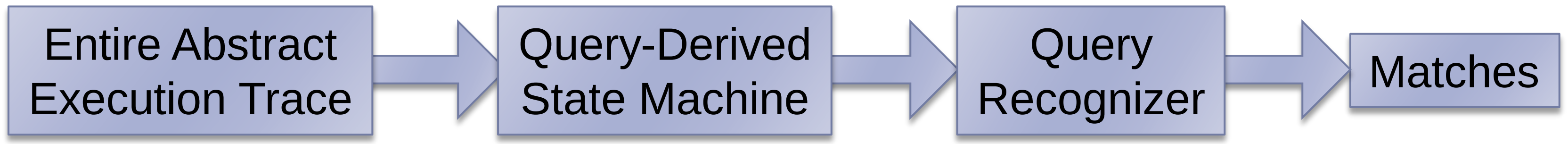
Graceful Match Handling with PQL

- Here is an example of how to trigger a function on a match in PQL:

```
query main()
uses String x, final;
matches {
    x = HttpServletRequest.getParameter(_)
  | x = HttpServletRequest.getHeader(_);
    final := derived(x);
}
replaces Connection.execute(final) with
    SQLUtil.safeExecute(x, final);
```

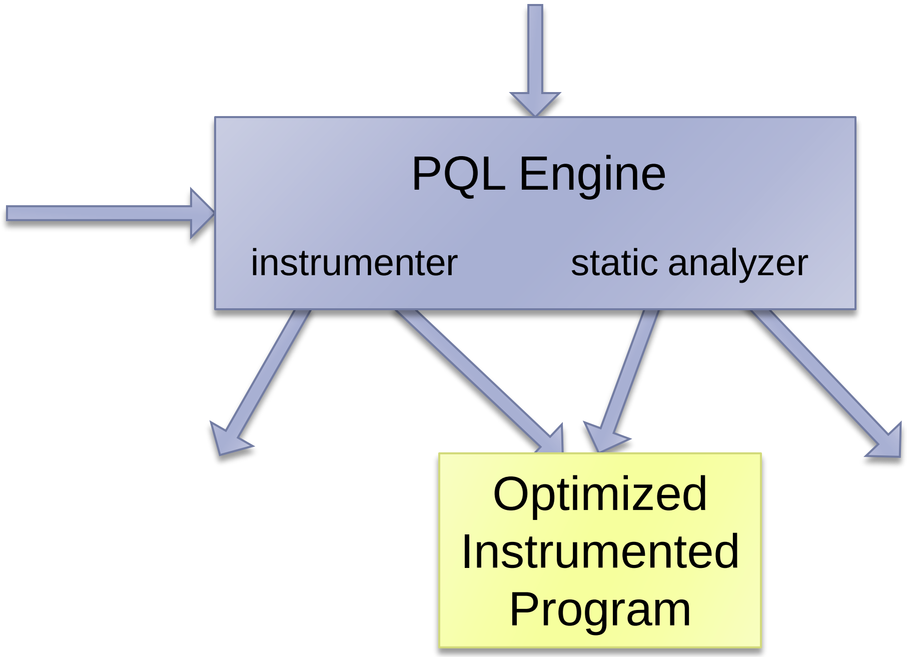
Going Deeper

- How does PQL do this dynamic analysis?
- A naïve dynamic analysis would translate the queries into state machines that would digest a program's full abstract execution trace:



More Efficient Dynamic Analysis

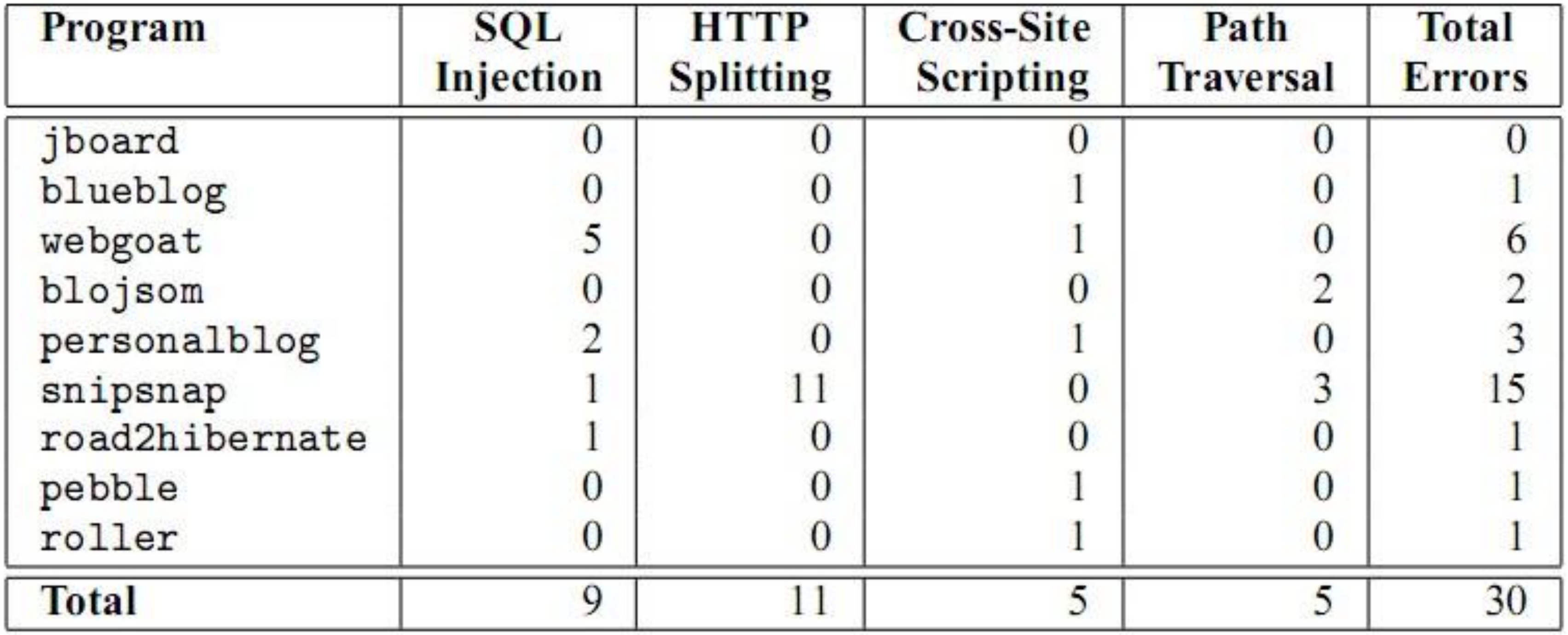
- Leverage results from static analysis
  - Reduce number of objects that need to be checked
  - Reduce number of points at which objects must be checked



A much more complete description of how PQL's dynamic analysis is done can be found here.

Experimental Results – Error Catching

PQL queries targeting different vulnerabilities were run on several open-source applications. The paper has a description of what each of these applications are for.



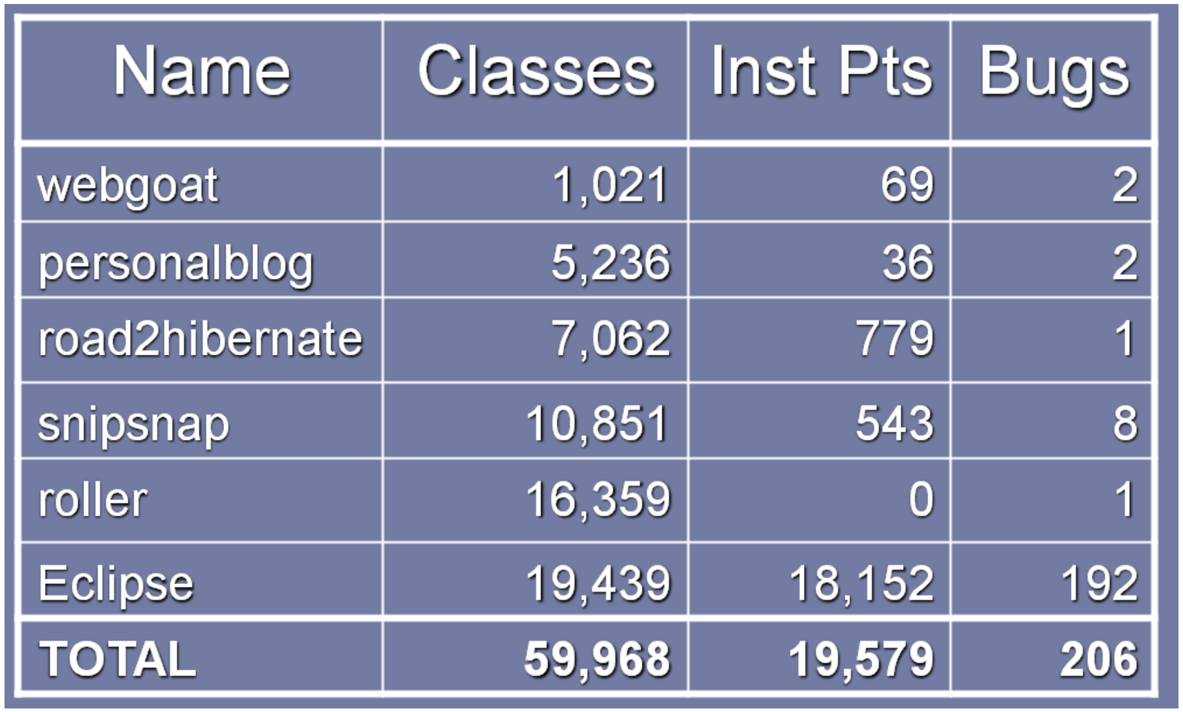
Program	SQL Injection	HTTP Splitting	Cross-Site Scripting	Path Traversal	Total Errors
jboard	0	0	0	0	0
blueblog	0	0	1	0	1
webgoat	5	0	1	0	6
blojsom	0	0	0	2	2
personalblog	2	0	1	0	3
snipsnap	1	11	0	3	15
road2hibernate	1	0	0	0	1
pebble	0	0	1	0	1
roller	0	0	1	0	1
Total	9	11	5	5	30

Experimental Results – False Positives

- The false positive rate was very high at first.
- Improvements to the underlying mechanics of the static analysis greatly decreased the number of false positives (from about 380 to 0 false positives in roller).
- More details about the improvements can be found in the Usenix ‘05 presentation slides here.

Experimental Results
PQL Dynamic Analysis Overhead

- Not a lot is mentioned about the overhead on a per-application basis.
- In general, the number of program points that have to be checked appears to be indicative of overhead.



Name	Classes	Inst Pts	Bugs
webgoat	1,021	69	2
personalblog	5,236	36	2
road2hibernate	7,062	779	1
snipsnap	10,851	543	8
roller	16,359	0	1
Eclipse	19,439	18,152	192
TOTAL	**59,968**	**19,579**	**206**

Related Work

- Static error-detection tools (SABER and WebSSARI are examples)
- Event-based Analysis
- Other Program Query Languages like ASTLOG and Jquery
- Analysis Generators

Benefits of PQL

- Division of labor between analysts and programmers.
  - Analysts can write PQL queries, programmers can run them and find bugs.
- The results indicate that it does help track down security bugs.
- The overhead of the dynamic analysis and protection is manageable enough that it can be run in real time.
- However… Humans still have to come up with the specifications. Wouldn’t it be great if there were a way to infer specifications from program code?

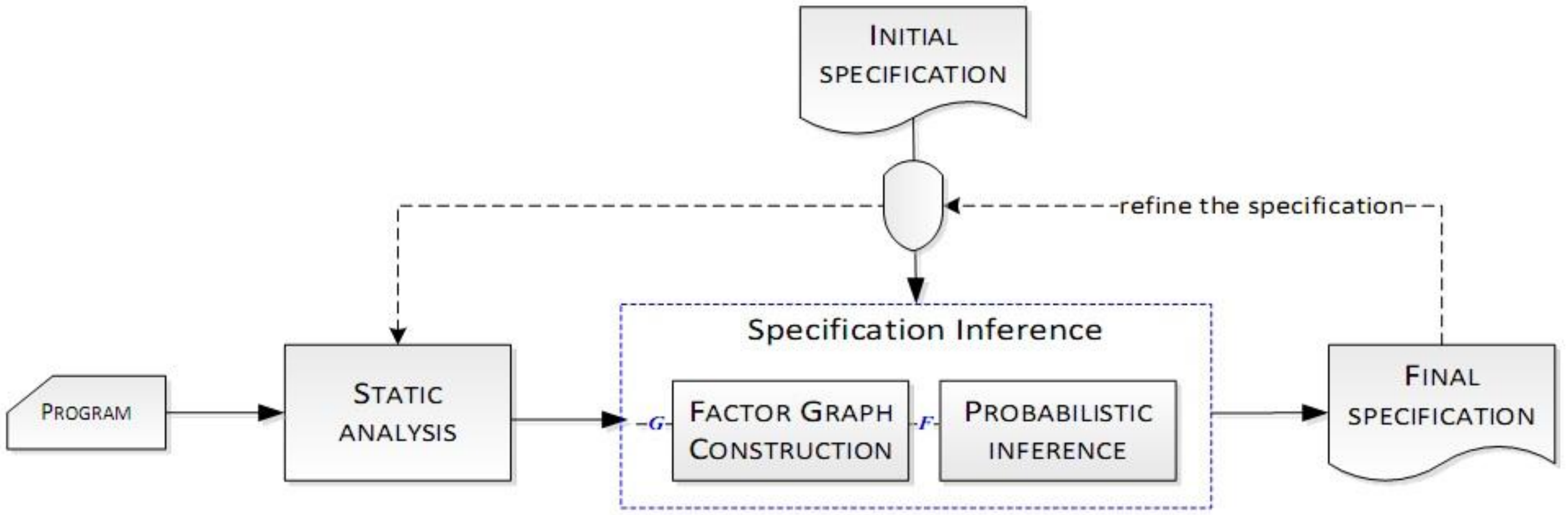
MERLIN

Specification Inference for

Explicit Information Flow Problems

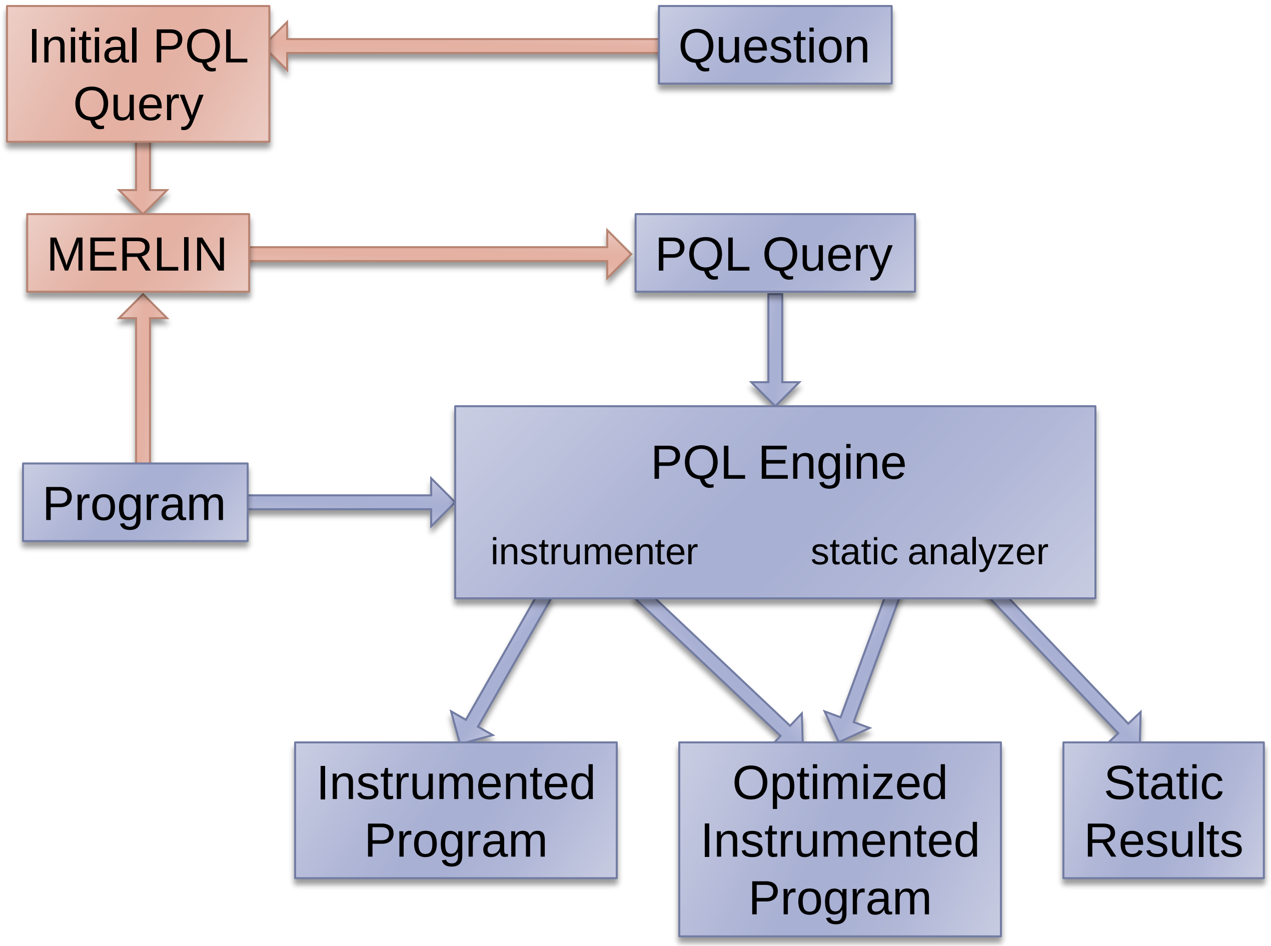
Overview of MERLIN

- Schematic of the MERLIN internal architecture:



- MERLIN can be integrated into the PQL architecture we have seen before.

MERLIN and the PQL Architecture



MERLIN Structure

- Probabilistic inference is the core of MERLIN.
- Used to solve a set of probabilistic constraints to generate a specification
- Relies on assumptions about the nature of propagation graphs for most programs.
  - Errors are rare.
  - There are only a few sanitizers.
- The assumptions can lead to contradictory constraints inferred from different paths.
  - Thus, constraints are parametrized with the probability of their satisfaction, based on a set of rules.
  - The MERLIN paper has much more information and can be found here.

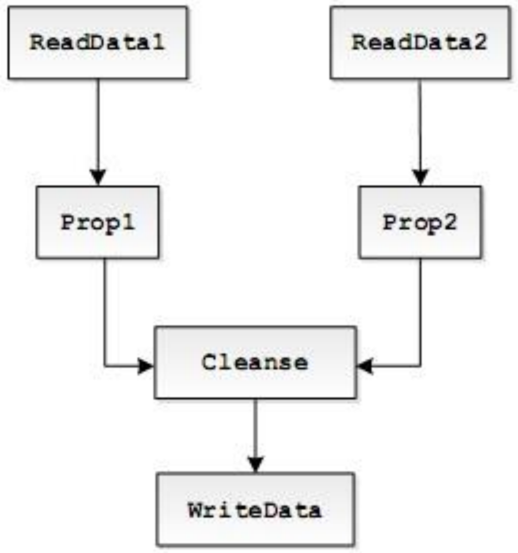
MERLIN Structure

- Control flow inside MERLIN starts with a propagation graph.
- Informally, propagation graphs describe the flow of information between methods.
- Generated by static analysis of the code.
- Approximate.
  - Pointer analysis is involved.
  - Cycles in the propagation graph are removed for simplification.

Propagation Graph Example

```
public void TestMethod1() {
  string a = ReadData1();
  string b = Prop1(a);
  string c = Cleanse(b);
  WriteData(c);
}

public void TestMethod2() {
  string d = ReadData2();
  string e = Prop2(d);
  string f = Cleanse(e);
  WriteData(f);
}
```



MERLIN Structure

- The construction of the factor graph and the probabilistic inference that follows is fairly involved.
- Key points:
  - Probabilistic constraints are generated from the propagation graph.
  - The conjunction of the constraints can be used to construct a factor graph.
  - The factor graph is used in the probabilistic inference to measure the odds that propagation graph nodes are sources, sanitizers, or sinks.

Experimental Results

- MERLIN is built as an add-on on top of CAT.NET, a publicly available static analysis tool.
- CAT.NET ships with an out-of-the-box specification, which MERLIN's results are compared to.
- 10 different benchmark programs used in test.
- MERLIN specifications:

GOOD	MAYBE	BAD
167	127	87

- This yields a false-positive rate of approximately 22%, which is quite decent.

Experimental Results (cont'd)

- When the good specifications generated by MERLIN were used to analyze the benchmark program code, fewer false positives were found.
- CAT.NET's false positive rate was 48% (43/89), MERLIN's false positive rate was 1% (3/342).
- MERLIN specifications also eliminated 13 former false positives.

Experimental Results (cont'd)

- CAT.NET + MERLIN Running Time

LoC	# DLLs	Time (s)
10,812	3	9.86
66,385	14	151.05
1,810,585	5	209.45

- The analysis seems to scale quite well, and the running times are fairly reasonable.

Summary

- PQL
  - Language to write abstract specifications in.
  - Allows for a division of labor between programmer and security analyst.
  - Includes the tools necessary to perform static and dynamic analyses using these specifications.
- MERLIN
  - Program to infer specifications automatically from program code.
- The combination of these two has a lot of potential.