Finding Application Errors and Security Flaws Using PQL: a Program Query Language

Michael Martin Benjamin Livshits Monica S. Lam
Computer Science Department
Stanford University
{mcmartin,livshits,lam}@cs.stanford.edu

ABSTRACT

A number of effective error detection tools have been built in recent years to check if a program conforms to certain design rules. An important class of design rules deals with sequences of events associated with a set of related objects. This paper presents a language called PQL (Program Query Language) that allows programmers to express such questions easily in an application-specific context. A query looks like a code excerpt corresponding to the shortest amount of code that would violate a design rule. Details of the target application's precise implementation are abstracted away. The programmer may also specify actions to perform when a match is found, such as recording relevant information or even correcting an erroneous execution on the fly.

We have developed both static and dynamic techniques to find solutions to PQL queries. Our static analyzer finds all potential matches conservatively using a context-sensitive, flow-insensitive, inclusion-based pointer alias analysis. Static results are also useful in reducing the number of instrumentation points for dynamic analysis. Our dynamic analyzer instruments the source program to catch all violations precisely as the program runs and to optionally perform user-specified actions.

We have implemented the techniques described in this paper and found 206 errors in 6 large real-world open-source Java applications containing a total of nearly 60,000 classes. These errors are important security flaws, resource leaks, and violations of consistency invariants. The combination of static and dynamic analysis proves effective at addressing a wide range of debugging and program comprehension queries. We have found that dynamic analysis is especially suitable for preventing errors such as security vulnerabilities at runtime.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging–Tracing

This work was supported in part by the National Science Foundation under Grant No. 0326227.

General Terms

Languages, Security, Reliability

Keywords

program traces, pattern matching, web applications, SQL injection, resource leaks, bug finding

1. INTRODUCTION

Advanced program analysis has been applied fruitfully to find large numbers of errors in software [10, 18, 20, 46, 54]. Program checkers are carefully crafted by experts and, as such, are targeted at finding patterns common to many application programs. In fact, these same techniques can also be used effectively to find error patterns that are specific to individual applications. To exploit the full potential of this approach, we need to make it easy for application developers to create their own custom checkers.

This paper presents Program Query Language (PQL), a language that allows developers to express a large class of application-specific code patterns. The system automatically generates from the query a pair of complementary checkers: a static checker that finds all potential matches in an application and a dynamic checker that traps all matches precisely as they occur, and can initiate user-specified logging or recovery actions upon a match. We have developed a prototype based on these ideas for Java, and used it to find and repair numerous security and resource management errors in large open-source applications.

1.1 A Simple Example

PQL focuses on the important class of error patterns that deal with sequences of events associated with a set of related objects. For example, for security reasons, a password received from the user should never be written out to disk without encryption. Such patterns are hard to express using conventional techniques like program assertions. The objects of interest may be stored in or passed between local variables, passed as parameters, or even passed through generic collections. The sequence of events may be scattered throughout many different methods and guarded by various predicates. PQL can express these kinds of patterns as the simplest prototypical code that exhibits the sequence of events of interest. The PQL system automatically finds all the matches in a program that have the equivalent behavior by abstracting away irrelevant control flow and disregarding how objects are named in the code.

As an example, let us consider SQL injection vulnerabilities [5, 27], ranked as one of the top five external threats to corporate IT systems [52]. Applications that use user-controlled input strings directly as database query commands are susceptible to SQL injections. Consider the following code fragment in a Java servlet hosting a Web service:

```
con.execute(request.getParameter("query"));
```

This code reads a parameter from an HTTP request and passes it directly to a database backend. By supplying an appropriate query, a malicious user can gain access to unauthorized data, damage the contents in the database, and in some cases, even execute arbitrary code on the server.

To catch this kind of vulnerability in applications, we wish to ask generally if there exist some

- object r of type `HttpServletRequest`,
- object c of type `Connection`, and
- object p of type `String`

in the code such that the result of invoking `getParameter` on r yields string p, and that string p is eventually used as a parameter to the invocation of `execute` on c. We can replace the call to `execute` with a custom routine `Util.CheckedSQL` that validates the query to ensure that it matches a permissible action. If the query is deemed invalid, the request is not made. Note that the two events in the application need not happen consecutively; the string p can be passed around as a parameter or stored on the heap before it is eventually used. We can express such a query in PQL as shown in Figure 1. The input to `r.getParameter` is immaterial, and is represented by the "don't care" symbol "_". Once the first call has been made, no additional matching is required except for trapping the final substitution. Note that SQL injections are more subtle in general and require more sophisticated patterns. The full SQL injection problem is discussed in Section 5.3.

Given an input pattern, the PQL system automatically produces a *sound* static checker that finds all its potential matches in a program. This checker uses a state-of-the-art context-sensitive inclusion-based points-to analysis [59]. The results are flow-insensitive with respect to the query; that is, for the query in Figure 1, the static checker would report all cases where the first argument of a call to `execute` is found to point to an object returned by some call to `getParameter`. The static checker does not ensure that the calls occur in the same order. This, combined with the intrinsic undecidability of statically matching a query for all possible runs, means that static results will generally have false positives. Note that, unlike many currently available static techniques, our static checkers are sound and will not produce false negatives; any possible match will be reported.

Dynamic checkers, on the other hand, can only find matches that actually occur at runtime, but are precise and permit online recovery actions to be triggered. When used as a dynamic tool, PQL creates an instrumented version of the input program that reports a runtime match if and only if there are object *instances* that match the query. For the query in Example 1, PQL will instrument the code to remember all object instances returned by `getParameter` and check them against the parameters supplied to `execute` routines. Should a match occur, it will intercept the call to `execute` and run the `Util.CheckSQL` routine instead.

```
query simpleSQLInjection()
uses
   object HttpServletRequest r;
   object Connection c;
   object String p;
matches { p = r.getParameter(_); }
replaces c.execute(p)
   with Util.CheckedSQL(c, p);
```

Figure 1: Simple SQL injection query.

Because the static results are sound and so have no false negatives, any point that the static analysis decides is irrelevant cannot possibly contribute to a match. The dynamic matcher is thus free to ignore it. PQL combines the two analyses by using the results of the static analysis to remove unnecessary instrumentation. In this example, only those `getParameter` and `execute` calls returned as potential matches by the static analysis need to be instrumented.

1.2 Contributions

PQL permits easy specification of a large class of patterns related to sequences of events on objects. A developer who needs to mine information from a program run can use it to produce targeted instrumentation. One who has just discovered a bug that he suspects also lurks elsewhere in the code can use it to quickly create a checker that will search for similar problems. To this end, PQL contributes the following:

Static checkers that leverage powerful analyses. An important result of this work is that we have placed sophisticated program analyses in the hands of developers. The developers can express simple queries and static checkers that use context-sensitive pointer alias analysis are automatically generated for them. Our analysis is sound, and as such its answer is guaranteed to include all points that may be relevant to the query. This allows its results to be used to optimize the dynamic matcher.

Optimized dynamic instrumentation. PQL automatically generates a specialized matcher for a query and weaves instrumentation into the target application to perform the match at runtime. These matchers differ from previous techniques in two ways. First, PQL transcends traditional syntax-based approaches by matching against the history of events witnessed by object instances. Second, the higher-level semantics of PQL make possible the use of static analysis to reduce the overhead of dynamic checking. Our system combines the dynamic matcher with sound static systems to produce its checkers.

Dynamic error recovery. PQL queries may specify functions to execute as a match is found, optionally replacing the last event with a user-specified function. This functionality can be used to recover from error conditions or to defend against attempts to breach application security.

Experimental evaluation of our approach. All the techniques in this paper have been implemented in a prototype system that works on Java programs. We have written tens of queries in the course of developing the tool and as part of our ongoing work. To explicitly test its ability to find bugs in large programs, we applied the technique to 6 large real-life applications with nearly 60,000 classes combined and found 206 errors. We found several security vulnerabilities and object persistence errors in Web applications that can permit database corruption or denial of service attacks. We also found resource management errors that eventually lead to memory exhaustion. The queries to find these errors were derived by reading descriptions of the error patterns and APIs of Java libraries, and by exploring the application code using PQL. The runtime overhead in these experiments ranged from 9% to 125% in the most heavily instrumented case. In the situations where overhead was the greatest, static analysis was applicable and removed between 82% to 99% of the instrumentation, and cut the overhead below 40%, and often below 3%. We also performed tests against standard benchmarks and found that in the extreme where nearly every event in significant, slowdown peaks at approximately 19 times.

Our experimental result suggests that the language covers an important class of program error patterns. Even though we were not familiar with the benchmarks, we were able to find errors in this large code base with relatively little effort.

1.3 Paper Organization

The rest of the paper is organized as follows. Section 2 gives an overview of the PQL language. Sections 3 and 4 describe our dynamic and static checkers, respectively. Section 5 provides a detailed experimental evaluation of our analysis approaches, while Section 6 discusses applications of our analyses in more general terms. Finally, Section 7 describes related work and Section 8 concludes.

2. PQL LANGUAGE OVERVIEW

The focus of PQL is to track method invocations and accesses of fields and array elements in related objects. To keep the language simple, PQL currently does not allow references to variables of primitive data types such as integers, floats and characters, nor primitive operations such as additions and multiplications. This is acceptable for object-oriented languages like Java because small methods are used to encapsulate most meaningful groups of primitive operations. The ability to match against primitive objects may be added to PQL as an extension in the future.

Conceptually, we model the dynamic program execution as a sequence of primitive events, in which the checkers find all subsequences that match the specified pattern. We first describe the abstract execution trace, then define the patterns describing subsequences of the trace.

2.1 Abstract Execution Traces

We abstract the program execution as a trace of primitive events, each of which contains a unique event ID, an event type, and a list of attributes. Objects are named by unique identifiers. PQL focuses on objects, and so it only matches against instructions that directly dereference objects. We also need to be able to detect the end of the program in order to match queries that demand that some other event never occur. As a result, all but the following eight event types are abstracted away:

- Field loads and stores. The attributes of these event types are the source object, target object, and the field name.
- Array loads and stores. The attributes of these event types are the source and target objects. The array index is ignored.
- Method calls and returns. The attributes of these event types are the method invoked, the formal objects passed in as arguments and the returned object. The return event parameter includes the ID of its corresponding call event.
- Object creations. The attributes of this event type are the newly returned object and its class.
- End of program. This event type has no attributes and occurs just before the Java Virtual Machine terminates.

Example 1. Abstract execution trace.
We illustrate the concept of an abstract execution trace with the code below:

```
int len = names.length;
for (int i = 0; i < len; i++) {
    String s = request.getParameter(names[i]);
    con.execute(s);
}
```

The code above runs through the array `names`; for each element, it reads in a parameter from the HTTP request and executes it. Figure 2 shows an abstract execution trace for the code in the case where the `names` array has two elements. Each event in the trace is listed with its ID, the ID of the caller in the case of a return, and

Event ID	Caller ID	Call/ Return	Event
1			$o_2 = o_1[\,]$
2		*call*	$o_4 = o_3$.getParameter(o_2)
3	2	*return*	$o_4 = o_3$.getParameter(o_2)
4		*call*	o_5.execute(o_4)
5	4	*return*	o_5.execute(o_4)
6			$o_6 = o_1[\,]$
7		*call*	$o_7 = o_3$.getParameter(o_6)
8	7	*return*	$o_7 = o_3$.getParameter(o_6)
9		*call*	o_5.execute(o_7)
10	9	*return*	o_5.execute(o_7)
11		*call*	o_5.execute(o_8)
12	11	*return*	o_5.execute(o_8)

Figure 2: Abstract execution trace for Example 1.

information on the event type and its attributes. In this execution, `names` is bound to object o_1; o_2 and o_6 are elements of array o_1 (the precise index is abstracted away); `request` is bound to object o_3, `s` is bound to object o_4 and o_7 in the first and second iteration, respectively.

This execution yields two matches to the `simpleSQLInjection` query in Figure 1. The first match is satisfied with $r = o_3$, $c = o_5$, $p = o_4$, and the second is satisfied with p matched to o_7, and the same values for r and c. □

2.2 PQL Queries

A PQL query is a pattern to be matched on the execution trace and actions to be performed upon the match. A match to the query is a set of objects and a subsequence of the trace that together satisfy the pattern.

The grammar of a PQL query is shown in Figure 3. The query execution pattern is specified with a set of primitive events connected by a number of constructs including sequencing, partial sequencing, and alternation. Named subqueries can be used to define recursive patterns. Primitive events are described using a Java-like syntax for readability. A query may declare typed variables, which will be matched against any values of that type and any of its subtypes. The use of the same query variable in multiple events indicates that the same object is used in all of the events.

Section 2.2.1 discusses variables in the context of a query, Section 2.2.2 outlines how statements are defined and combined, Section 2.2.3 describes PQL's subquery mechanism, and Section 2.2.4 discusses the options PQL provides for reacting to a match.

2.2.1 Query Variables

Query variables correspond to objects in the program that are relevant to a match. They are declared inside of subqueries and are local to the query they are declared in.

The most common variables represent *objects*, and represent individual objects on the heap, Object variables have a class name that restricts the kind of object instances that they can match. If that name is prefixed with a "!", then the object must *not* be castable to that type. If the same object variable appears multiple times in a query, it must be matched to the same object instance. The *contents* of the object need not be the same for multiple matches.

```
queries       ⟶  query*

query         ⟶  query qid ( [decl [, decl]*] )
                 [returns declList ; ]
                 [uses declList ; ]
                 [within methodInvoc )]
                 [matches { seqStmt }]
                 [replaces primStmt with methodInvoc ;]*
                 [executes methodInvoc [, methodInvoc]* ;]*

methodInvoc   ⟶  methodName(idList)

decl          ⟶  object [!] typeName id |
                 member namePattern id
declList      ⟶  object [!] typeName id ( , id )* |
                 member namePattern id ( , id )*
stmt          ⟶  primStmt | ∼ primStmt |
                 unifyStmt | { seqStmt }

primStmt      ⟶  fieldAccess = id |
                 id = fieldAccess |
                 id [ ] = id |
                 id = id [ ] |
                 id = methodName ( idList ) |
                 id = new typeName ( idList )


seqStmt       ⟶  ( poStmt ; )*
poStmt        ⟶  altStmt ( , altStmt )*
altStmt       ⟶  stmt ( "|" stmt )*

unifyStmt     ⟶  id := id
                 ( [idList] ) := qid ( idList )

typeName      ⟶  id ( . id )*
idList        ⟶  [ id ( , id )* ]
fieldAccess   ⟶  id . id
methodName    ⟶  typeName . id
id,qid        ⟶  [A-Za-z_][0-9A-Za-z_ ]*
namePattern   ⟶  [A-Za-z*_ ][0-9A-Za-z*_ ]*
```

Figure 3: BNF grammar specification for PQL.

There are also *member* variables, which represent the name of a field or a method. Member variables are declared with textual pattern that the member name must match. A pattern of "$*$" will match any method name. If a member variable occurs multiple times in a pattern, it must represent the same field or method name in each event.

For convenience, we introduce a wildcard symbol "_" whose different occurrences can be matched to different member names or objects. However, values matched to wildcard symbols cannot be examined or returned.

Query variables are either *arguments* (passed in from some other query that has invoked it), *return values* (acted upon by the query's action, or returned to an invoking query, or both), or *internal variables* (used inside the query to find a match, but otherwise isolated from the rest of the system).

2.2.2 Statements

Most primitive statements in our query language correspond directly to the event types of the abstract execution trace. Method invocations are the exception to this; they match all events between a call to the method and its matching return event. References to objects in a primitive statement must be declared object query variables, or the special variable "_", which is a wildcard placeholder for any object not relevant to the query. References to members may be literals or declared member query variables. A field or method in an event need not be declared in the type associated with its base variable; in such cases, a match can only occur if a subclass defines it.

Primitive statements may be combined into compound statements, as shown in the grammar. A sequence $a; b$ specifies that a is followed by b. Ordinarily, this means any events may occur between them as well—the primary focus is on individual objects, so sequences are, by default, not contiguous. An event may be forbidden from occurring at a point in the match by prefixing it with the exclusion operator "$\sim$". Thus, the sequence $a; \sim b; c$ matches a followed by c if and only if b does not occur between them. Wildcards are permissible, so excluding all possible events can force a sequence to be contiguous in the trace if desired.

The alternation operator is used when we wish to match any of several events (or compound statements): if a and b are statements, then $a|b$ is the statement matching either a or b.

To match multiple statements independently of one another, we use partial-order statements, which separate the statements to be matched with commas. The statement $a, b, c;$ would match the three statements a, b, and c in any order. If a clause in a partial-order statement is a sequence itself, then sequencing within that clause is enforced as normal.

Of the three combination operators, alternation has the highest precedence, then partial-order, and lastly sequencing. Braces may be used to enforce the desired precedence.

The `within` construct is introduced to allow the specification of a pattern to *fully* match within a (dynamic) invocation of a method. This translates to matching against a method call event, then matching the pattern—and insisting that the return of the method not occur at any point between the call and the full match of the pattern.

Queries that end with excluded events represent *liveness properties*. If the query is embedded in a `within` clause, then it will return a match if and when the end of the invocation of the method is reached without the excluded event occurring. If the main query ends with excluded events, then the match cannot be confirmed until the program exits.

Example 2. Forcing closing of stream resources.
Java has many automatic resource management features, but system-wide resources such as file handles still must be manually released or the system risks resource exhaustion. Some methodologies demand that resources allocated in a method must be deallocated before the method ends [58].

```
query forceClose()
uses object InputStream in;
within _ . _ ();
matches {
    in = new InputStream();
    ∼in.close();
}
executes in.close();
```

Figure 4: Checking for potential leaks of file handles.

Shown in Figure 4 is a query that finds all methods that do not manage `InputStream` resources according to such a methodology. Here, a match is found if some invocation to method m creates an `InputStream in`, and that `in` is not closed before the method ends. Should it escape the allocating method, a call to `close` is inserted. Note that `close` need not be invoked directly by m; it can be invoked by a method called by m. Ordinarily, the `within` clause specifies a particular method of interest; for this problem, the pattern applies to all methods, and so both the base object and the method name are wildcards. □

2.2.3 Subqueries

Subqueries allow users to specify recursive event sequences or recursive object relations. Subqueries are defined in a manner analogous to functions in a programming language. They can return multiple values, which are bound to variables in the calling query. By recursively invoking subqueries, each with its own set of variables, queries can match against an unbounded number of objects.

Values from input and return query variables are transferred across subqueries by unifying formals with actuals, and return values with the caller's variables. *Unification* in the context of a PQL match involves ensuring that the two unified variables are bound to the same value in any match. If one variable has been bound by a previous event but the other has not, the undefined variable is bound to the same value. If both have already been bound to different variables, then no match is possible.

When writing recursive subqueries, it is often necessary for the base case to force the return value to be equal to one of its arguments. PQL provides a unification statement to express this: the statement `a` := `b` does not correspond to any program event, but instead unifies its parameters `a` and `b`.

Example 3. Recursive subqueries.

Recursion is useful for matching against the common idiom of wrappers in Java. Java exposes higher-level I/O functions by providing wrappers over base input streams. These wrappers are subclasses of the top-level interfaces `Reader` (for character streams) and `InputStream` (for byte streams). For example, to read Java Objects from some socket `s`, one might first wrap the stream with a `BufferedInputStream` to cache incoming data, then with an `ObjectInputStream` to parse the objects from the stream:

```
r1 = new BufferedInputStream(s.getInputStream()));
r2 = new ObjectInputStream(r1);
obj = r2.readObject();
```

In general, there can be arbitrary levels of wrapping. To capture this, we need to use a recursive pattern, as shown in Figure 5. The base case in *derivedStream* subquery declares that any stream can be considered derived from itself; the other captures a single wrapper and then re-invokes *derivedStream* recursively. □

```
query derivedStream(object InputStream x)
returns object InputStream d;
uses object InputStream t;
matches {
    d := x
    | {t = new InputStream(x);
      d := derivedStream(tmp);}
}

query main()
returns method * m;
uses
    object Socket s;
    object InputStream x, y;
    object Object v;
matches {
    x = s.getInputStream();
    y := derivedStream(x);
    v = y.readObject();
    v.m();
}
executes Util.PrintStackTrace(*);
```

Figure 5: Recursive query for tracking data from sockets.

It is natural to ask how a developer should go about writing a PQL query. In some cases such as SQL injections, studying the API of relevant methods is sufficient. Sometimes it is useful to instead locate objects of interest and see how these objects are used in the program. PQL makes such explorations easy. For example, the query shown in Figure 5 finds all methods invoked on objects read from a network socket. The query first finds all the streams derived from the input stream of a socket, then all objects read from any of the derived streams. It then matches against any method, represented by the method parameter `m`, invoked upon the objects read.

2.2.4 Reacting to a Match

Matches in PQL often correspond to notable or undesirable program behavior. PQL provides two facilities to log information about matches or perform recovery actions.

The simplest version of these is the `executes` clause, which names a method to run once the query matches. PQL subqueries may also have one or more `replaces` clauses. These name a statement to watch for, and a method representing the action to be executed in its place. This method may take query variables as arguments. Passing the special symbol "∗" as an argument will package every variable binding in the match into a collection that can be handled generically.

Some basic actions are defined as methods in a class `Util` as part of the base system; the two most frequently used are `Util.PrintStackTrace`, which takes the "∗" argument and dumps information about the variable values and the stack trace where the final event occurred, and `Util.Abort`, which takes no arguments and terminates the program immediately. Both are intended for the `executes` clause.

When implementing actions, the method must return `void` for `executes` clauses, or a value of the same type of the replaced event for `replaces`. Each argument to the action is represented as an array of Objects. Arrays are necessary because multiple matches may complete on a single event. Each index into the argument array corresponds to a single match that has completed.

2.3 Expressiveness of PQL

PQL as a pattern language is fundamentally concerned with objects. It seeks to find a set of heap objects (disregarding how they are named syntactically in the code) to parameterize a context-sensitive pattern of events over the execution trace.

The events in these patterns do not refer to primitive values such as integers or individual characters, and so PQL is not capable of tracking them.

The subquery mechanism introduces a call chain that permits the matcher to match context-free grammars. Each production in such a grammar can be considered to be independently existentially quantified with respect to objects on the heap. This means that, despite the fact that any query can only refer to a finite number of variables, any number of objects may be involved in a match on a recursive query.

PQL does not directly provide a Kleene-star operator. However, this facility may be simulated with tail-recursive queries. In practice, useful queries with loops need to refer to different objects or chains of objects, and in both cases, a simple Kleene star is insufficient to capture the precise semantics.

The partial-order operator specifies that the execution stream must be able to match each of several clauses, which is equivalent to specifying the intersection of the languages specified by each clause. PQL's class of languages is thus that of the closure of context-free languages combined with intersection; this class is a superset of context-free languages.

The default semantics of the sequencing operator in PQL would also seem to require patterns to be unduly permissive, since they permit arbitrary blocks of statements to occur between events of interest. However, due to the object-centric focus of the language, this usually is precisely the behavior desired. For the occasions when this is not what is desired, one can use exclusion events to forbid all intervening events. This permits the language to express arbitrary patterns on the execution trace, while keeping the most generally useful patterns the simplest to express.

3. DYNAMIC MATCHER

A direct, naïve approach to finding matches to PQL queries dynamically consists of the following three steps:

1. Translate each subquery into a non-deterministic state machine which takes an input event sequence, finds subsequences that match the query and reports the values bound to all the returned query variables for each match. If there is only a `main` query, this can be a simple finite state machine. More complicated queries require additional machinery, described in Section 3.1.
2. Instrument the target application to produce the full abstract execution trace.
3. Use a query recognizer to interpret all the state machines over the execution trace to find all matches.

The procedure as described is quite inefficient. To reduce instrumentation overhead, we perform the following optimizations. First, instrumentation code is inserted only at those program points that might generate an event of interest for the specific query. A simple type analysis excludes operations on types not related to objects in the query. We use the results of our static analysis, described in Section 4, to further reduce the instrumentation by excluding statements that cannot refer to objects involved in any match of the query. Also, instead of collecting full traces, our system tracks all the partial matches as the program executes and takes action immediately upon recognizing a match.

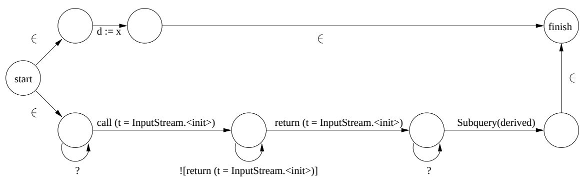
3.1 Translation From Queries To State Machines

A state machine representing a query is composed of: a set of states, which includes a start state, a fail state, and an accept state; a set of state transitions; and a set of variable parameters. A partial match is given by a current state and a set of *bindings*—a mapping from variables in a PQL query to objects in the heap at run time. A state transition specifies the event for which under which a current state and current bindings transition to the next state and a new set of bindings. Because the same event may be interpreted in different ways by different transitions, a state machine may non-deterministically transition to different states given the same input.

To represent partial-order statements, states can be made to be *join points*: in these cases, the state machine must determine that every incoming transition has a compatible partial match. Each outgoing match from such a state is a combination of one incoming match from each transition; every possible consistent combination is formed. A combination is consistent if and only if no two matches define the same variable to different values. In the extreme case where no incoming transition binds any variable bound by any other transition, the outgoing matches are the cartesian product of all incoming matches.

State transitions generally represent a single primitive statement corresponding to a single event in the execution trace. There are three special kinds of transitions:

Skip transitions. A query specifies a *subsequence* of events to match. Unless noted otherwise with an exclusion statement, an arbitrary number of events of any kind are allowed in between con-



start
d := x
finish
ϵ
ϵ
ϵ
ϵ
call (t = InputStream.<init>)
return (t = InputStream.<init>)
Subquery(derived)
?
![return (t = InputStream.<init>)]
?

Figure 6: State machine for the `derivedStream` query.

secutive matched statements. We represent this notion with a *skip transition*, which connects a state back to itself on any event that doesn't match the set of excluded events. Note that the accept state does not have a skip transition, so matches are reported only once. We label skip transitions with a "?" to indicate that they may match any event, or with "![event]" to indicate that the transition will match any event but the one listed.

Null transitions. A null (ϵ) transition does not correspond to any event; it is taken immediately when encountered. Any state with outgoing ϵ transitions must have all outgoing transitions be ϵ. They may optionally carry a predicate; the transition may only be taken if the predicate is true. If it is not, the matcher transitions directly into the fail state.

Subquery invocation transitions. These behave mostly like ordinary transitions, but correspond to the matches of entire, possibly recursive, queries.

We preprocess the queries to ease translation. No subquery may, directly or indirectly, invoke itself without any intervening events. So, first we eliminate such situations, a process analogous to the elimination of left-recursion from a context-free grammar [1]. Second, excluded events are propagated forward through subquery calls and returns so that each set of excluded events is either at the end of `main` or immediately before a primitive statement.

We now present a syntax-directed approach to constructing the state machine for a query. Associated with each statement s in the query are two states, denoted $bef(s)$ and $aft(s)$, to refer to the states just before and after s is matched. For a query with statement s, the start and accept states of the query are states $bef(s)$ and $aft(s)$, respectively. As an example, the `derivedStream` query from Figure 5 is translated to the state machine shown in Figure 6.

Array and field operations. These are the primitive statements that correspond to single events in the trace. For a primitive statement s of type t, the transition from $bef(s)$ to $aft(s)$ is predicated by getting an input event also of type t and that the attributes in e must be *unifiable* with those in statement s and the current bindings. An attribute in e with value x is unifiable with s and current bindings if

1. either the corresponding attribute in s has a literal value x
2. or it refers to a parameter variable v that is either unbound or bound to value x.

If the attribute refers to an unbound variable v, the pair (v, x) is added to the set of known bindings.

Exclusion. For an excluded primitive statement of the form $\sim s'$, $bef(s) = aft(s)$. The default skip transition is modified to be predicated upon not matching s'.

Sequencing. If $s = s_1; s_2$, then $bef(s) = bef(s_1)$, $aft(s) = aft(s_2)$, and $aft(s_1) = bef(s_2)$.

Alternation. If $s = s_1 | s_2$, then $bef(s)$ provides ϵ transitions to $bef(s_1)$ and $bef(s_2)$; similarly, $aft(s_1)$ and $aft(s_2)$ each have an ϵ transition to $aft(s)$.

Partial order. Partial orders resemble alternation statements: if $s = s_1, s_2$, then $bef(s)$ provides ϵ transitions to $bef(s_1)$ and

$bef(s_2)$; similarly, $aft(s_1)$ and $aft(s_2)$ each have an ϵ transition to $aft(s)$. The primary difference is that the $aft(s)$ state is a join point.

Method invocation. If s is a method invocation statement, we must match the call and return events for that method, as well as all events between them. To do this, we create a fresh state t and a new event variable v. We create a transition from $bef(s)$ to t that matches the `call` event, and bind v to the ID of the event. We create another transition from t to $aft(s)$ that matches a `return` event with ID v. The skip transition from t back to itself is modified to exclude the match of the return event. Call and return events are unified in a manner analogous to array and field operations.

Creation points. Object creation is handled in Java by invoking the method "$<$ `init` $>$", and is translated like any other method invocation.

Context. The *within* clause is represented by nesting the automaton representing the body between a pair of matching call and return event pairs. The skip transitions are modified to not match the return, forcing the failure of any match that does not complete within the call.

Unification statements. A unification statement is represented by a predicated ϵ transition that requires that the two variables on the left and right have the same value. If one is unbound, it will acquire the value of the other.

Subquery invocation. Subquery invocations are treated as if it the subquery match were a primitive event in its own right. The recognizer handles subquery calls and returns on its own. More details are discussed in Section 3.3.

3.2 Instrumenting the Application

The system instruments all instructions in the target application that match any primitive event or any exclusion event in the query. At an instrumentation point, the pending event and all relevant objects are marshalled and sent to the query recognizer. The recognizer will update the state of all pending matches and then return control to the application.

The recognizer does not interfere with the behavior of the application except via completed matches; therefore, any instrumentation point that can be statically proven to not contribute to any match need not be instrumented. In particular, we can optimize away instrumentation where the referenced objects have statically declared types that conflict with the query. More sophisticated optimization techniques are discussed in Section 4.

3.3 The Query Recognizer

The recognizer begins with a single partial match at the beginning of the main query, with no values for any variables. It receives events from the instrumented application and updates all currently active partial matches. For each partial match, each transition from its current state that can unify with the event produces a new possible partial match where that transition is taken. A single event may be unifiable with multiple transitions from a state, so multiple new partial matches are possible. If a skip transition is present and its predicates pass, the match will persist unchanged. If the skip transition is present but a predicate fails the match transitions to the fail state. If the skip transition is present but a predicate's value is unknown because the variables it refers to as are of yet unbound, then the variable is bound to a value representing "any object that does not violate the predicate." Predicates accumulate if two such objects are unified; unification with any object that satisfies all such predicates replaces the predicates with that object.

If the new state has ϵ transitions, they are processed immediately. If a transition representing a subquery call is available from the new state, a new partial match based on the subquery's state machine is generated. This partial match begins in the subquery's start state and has initial bindings corresponding to the arguments the subquery was invoked with. A unique subquery ID is generated for the subquery call and associated with the subquery caller's partial match, with the subquery callee's partial match, and with any partial match that results from taking transitions within the subquery callee.

Join points are handled by finding the latest state that dominates the join point and treating it as the *split point*. Each incoming transition to the join point has a submachine representing all paths from the split point to that transition. When a split point is reached, each of these submachines is matched independently in a manner similar to subqueries. The join point then collects and combines matches as they complete, and propagates combined matches once all of them have completed.

Once a partial match transitions into an accept state, it begins to wait for events named in `replaces` clauses. When a targeted event is encountered, the instruction is skipped and the substituted method is run instead. An `executes` clause runs immediately once the accept state is reached.

When a subquery invocation completes, the subquery ID is used to locate the transition that triggered the subquery invocation. The variables assigned by the query invocation are then unified with the return values, and the subquery invocation transition is completed. The original calling partial match remains active to accept any additional subquery matches that may occur later.

In order for this matcher to scale over long input traces, it is critical to be able to quickly acquire all relevant partial matches to an event. We use a hash map to quickly access partial matches affected by each kind of event. This map is keyed not only on the specific transition, but also on all variables known to have values at that point in the query. For queries whose partial matches consist of at most one variable-value pair of binding, our implementation is very efficient as it needs to perform only one single hash lookup.

4. STATIC CHECKER & OPTIMIZER

PQL makes it easy for developers to take advantage of context-sensitive points-to analysis results. We have developed an algorithm to automatically translate PQL queries into queries on a pointer analysis result, shielding the user from the need to directly operate on the program representation or the context-sensitive results. This translation approach is very flexible: even though our checkers are currently flow-insensitive, flow sensitivity can be added in the future to improve precision without needing to modify the queries themselves.

Accurate interprocedural pointer alias analysis is critical to the precision of PQL static checkers, because events relevant for a particular query may be widely separated in the program. The analysis for PQL must be sound, because false negatives mean the results are unusable for optimization. This is in contrast to many recently developed practical static checkers [10, 18, 62] which use unsound analyses and thus produce both false-negative and false-positive warnings.

Our checkers use pointer information from a sound cloning-based context-sensitive inclusion-based pointer alias analysis due to Whaley and Lam [59]. This analysis computes the points-to relations for each distinct call path for programs without recursion. Call paths in recursive programs are reduced by treating each strongly connected component as a single node. The points-to information is stored in a deductive database called `bddbddb`. The data are compactly represented with binary decision diagrams

(BDDs), and can be accessed efficiently with queries written in the logic programming language Datalog.

4.1 The bddbddb Program Database

All inputs and results for the static analyzer are stored as relations in the bddbddb database. The domains in the database include bytecodes B, variables V, methods M, contexts C, heap objects named by their allocation site H, and integers Z. The context domain represents the various call chains that can occur in the program, and is used to qualify pointer information. Two pointer relations that are true in the same calling context are associated with the same value in C. For a further treatment of this, see [59].

The source program is represented as a number of input relations: $actual, ret, fldld, fldst, arrayld, arrayst$ represent parameter passing, method returns, field loads, field stores, array loads, and array stores, respectively. There is a one-to-one correspondence between attributes of primitive statements in the query language and those in the relations.

In the following, we say that predicate $A(x_1, \ldots, x_n)$ is true if tuple $(x_1, \ldots, x_n)$ is in relation A. Below we show the definitions of three of the relations; the remaining ones are defined similarly.

$fldld$: $B \times V \times M \times V$. $fldld(b, v_1, m, v_2)$, means that bytecode b executes "$v_1 = v_2.m$".

$actual$: $B \times Z \times V$. $actual(b, z, v)$ means that variable v is zth argument of the method call at bytecode b.

ret: $B \times V$. $ret(b, v)$, means that variable v is the return result of the method call at bytecode b.

The context-sensitive points-to analysis produces a numbering of the calling contexts, the invocation graph of the context-sensitive call graph, and finally the points-to results:

IE: $C \times B \times C \times M$ is the context-sensitive invocation relation. $IE(c_1, i, c_2, m)$ means that invocation site i in context c_1 may invoke method m in context c_2.

vP: $C \times V \times H$ is the variable points-to relation. $vP(c, v, h)$ means that variable v in context c may point to heap object h.

A Datalog query consists of a set of rules, written in a Prolog-style notation, where a predicate is defined as a conjunction of other predicates. For example, the Datalog rule

$$D(w, z) \quad :- \quad A(w, x), B(x, y), C(y, z).$$

says that "$D(w, z)$ is true if $A(w, x)$, $B(x, y)$, and $C(y, z)$ are all true."

Example 4. Statically detecting basic SQL injections.

We can express a flow-insensitive approximation of the basic SQL injection query in Figure 1 as follows:

$$\begin{aligned}
&simpleSQLInjection(b_1, b_2, h) :- \\
&\qquad IE(c_1, b_1, \_, \text{“}\texttt{getParameter}\text{”}), \\
&\qquad ret(b_1, v_1), vP(c_1, v_1, h), \\
&\qquad IE(c_2, b_2, \_, \text{“}\texttt{execute}\text{”}), \\
&\qquad actual(b_2, 1, v_2), vP(c_2, v_2, h).
\end{aligned}$$

The Datalog rule says that an object h is a cause of an injection if b_1 is a call to `getParameter`, b_2 is a call of `execute`, and the return result of `getParameter` v_1 in some context c_1 points to the same heap object h as v_2, the first parameter of the call to `execute` in some context c_2. □

4.2 Translation from PQL to Datalog

We perform static analysis by translating PQL queries into Datalog and using bddbddb to resolve the queries. Datalog is a highly expressive language, including the ability to recursively specify properties, meaning that PQL queries may be translated to Datalog approximation using a simple syntax-directed approach.

In the beginning of the translation process, we *normalize* the input PQL queries so that the `matches` part of each query is an alternation of sequence statements; in other words, the top most level statement of the `matches` clause is an *altStmt* each of which clauses is a *seqStmt* and *altStmt*s mentioned in Figure 3 are used *only* at the top level. Any event affected by a `replaces` clause is treated by this process as being a possible final event in the query. This is equivalent to appending an alternation of all such statements to the end of the `matches` clause before normalization.

Each PQL query becomes a Datalog relation defined over bytecodes, field/method names, and heap variables; one bytecode for every program point in the longest possible sequence of events through the query, one field or method name for each member variable in the PQL query, and one heap variable for each object variable in the PQL query. Literals and wildcards are translated from PQL into Datalog without change. We summarize the handling of individual PQL constructs below:

Primitive statements. Each primitive statement in the query is translated into one or more Datalog predicates. A syntax-directed translation of PQL queries into Datalog is shown in Figure 7. The left side of the table lists a PQL primitive statement and the right hand side shows its Datalog translation. All of these translations have the same basic form. The PQL statement refers to some heap object h_i. The bddbddb system, however, represents instructions in terms of actual program variables. We must therefore first extract the program variables into some fresh Datalog variable v_{hi} and then query the vP relation to determine the possible values for h_i. If a field or method name refers to a PQL member variable, it may be referenced directly in the statement.

Alternation. Since the input queries are normalized so that alternation statements are used only at the top level, each clause in an alternative is represented by a separate Datalog rule with the same head goal.

Sequencing. Because the static analysis is flow-insensitive, we do not track sequencing directly, and instead merely demand that all events in the sequence occur at some point. This can be done by simply replacing the sequence operator ";" with the Datalog conjunction operator ",". Since there is no guarantee that the same program variables are used in each event in the sequence, the v_x, i, and c Datalog variables must be fresh for each event. The h_x and m variables correspond to PQL constructs and so keep the same name. Each event includes a reference to a bytecode where the event occurs, and this bytecode is bound to one of the bytecode attributes of the subquery relation. If bytecode parameters are left unbound (for example, in the base case for the `derivedStream` query in Figure 5, the unused bytecode parameters are set to NULL, representing no program location.

Partial order. Similarly, because the static analysis is flow-insensitive, the translation of a partial-order statement can simply treat all its clauses as part of a sequence.

Exclusion. With flow-insensitivity, no guarantees about ordering can be made. This means that we cannot deduce that an excluded event (denoted with a $\sim$) occurs between two points in a sequence; as a result, all excluded events are ignored. This is a source of imprecision in our current analysis, but it is a conservative approximation that maintains soundness.

Primitive Statement	Datalog translation
primStmt $\longrightarrow$ $h_1.m = h_2$	$\mathit{fldst}(\_, v_1, m, v_2)$, $vP(c, v_1, h_1)$, $vP(c, v_2, h_2)$
primStmt $\longrightarrow$ $h_1 = h_2.m$	$\mathit{fldld}(\_, v_1, m, v_2)$, $vP(c, v_2, h_1)$, $vP(c, v_1, h_2)$
primStmt $\longrightarrow$ $h_1[\,] = h_2$	$\mathit{arrayst}(\_, v_1, v_2)$, $vP(c, v_1, h_1)$, $vP(c, v_2, h_2)$
primStmt $\longrightarrow$ $h_1 = h_2[\,]$	$\mathit{arrayld}(\_, v_1, v_2)$, $vP(c, v_2, h_1)$, $vP(c, v_1, h_2)$
primStmt $\longrightarrow$ h_0 = m ($h_1, \ldots, h_n$)	$\mathit{ret}(i, h_0), \mathit{IE}(c, i, \_, m)$, $\mathit{actual}(i, 1, v_1), vP(c, v_1, h_1)$, $\ldots$ $\mathit{actual}(i, n, v_n), vP(c, v_n, h_n)$
primStmt $\longrightarrow$ h_0 = **new** *typeName* ($h_1, \ldots, h_n$)	$\mathit{ret}(i, h_0), \mathit{IE}(c, i, \_, \mathit{typeName}.\texttt{<init>})$, $\mathit{actual}(i, 1, v_1), vP(c, v_1, h_1)$, $\ldots$ $\mathit{actual}(i, n, v_n), vP(c, v_n, h_n)$

Figure 7: Translation of primitive statements $\mathit{primStmt}$ from PQL (left) into Datalog (right) for static analysis and optimization.

Within. The `within` m construct is handled by requiring that matching bytecodes be found in methods transitively called from m. This involves querying a call graph; such a call graph is available as part of the pointer analysis.

Unification. Unification of objects is translated into equality of heap allocation sites.

Subqueries. Invocations of PQL subqueries are represented by referring to the equivalent Datalog relation. The program points and any variables that are not parameters in the PQL subquery are matched to wildcards and projected away.

Figure 8 shows a full translation of the query from Figure 5 into Datalog. The first rule is a translation of the `main` query, which has only one path. It also involves three events, one member variable, and four object variables. Each of the four PQL statements is translated in turn, and combined they form the core of the main query. The `derivedStream` query is somewhat more interesting, as it has two possible sequences, each with a different length. The base case is given first, which simply asserts the equality of its arguments (the unification statement) and then returns immediately. As there is no event in this path, the bytecode argument for `derivedStream` is set to NULL. The second rule handles the recursive case and is similar to `main`'s translation.

The $\mathit{mainRelevant}$ and $\mathit{derivedRelevant}$ relations express which bytecodes are actually part of the final solution, and will be explained in detail at the end of the next section.

4.3 Extracting the Relevant Bytecodes

The bddbddb system resolves each query more or less independently; as a result, each subquery finds program points and heap variables for any set of arguments, regardless of whether or not the subquery can be invoked with those arguments. It is thus necessary, when extracting the list of relevant bytecodes, to extract only those bytecodes that actually participate in a match of the full query. This is a two-step process. In the first step, we determine which subquery invocations contribute to the final result; in the second we project the relevant subqueries onto the bytecode domain.

Finding relevant subquery matches. Relevant subqueries are determined inductively: All members of the `main` relation is relevant, and any member of any query relation that appears as a clause in a relevant relation as the result of translating a subquery invocation is relevant. This translates to one rule for each invocation statement and an additional rule to express that all results of `main` are relevant. In Figure 8, the single rule for $\mathit{mainRelevant}$ declares any solution to main to be relevant. There are two invocations of the $\mathit{derivedStream}$ subquery, and each gets its own rule. The first handles the recursive subquery inside $\mathit{derivedStream}$, and the second deals with the call from main.

Extracting relevant program locations. Gathering the relevant program locations is straightforward once the previous step is performed; any program location that occurs in a relevant solution to any query is relevant. For the special case of the main query, we need not check for relevance because all solutions to the main query are relevant. Figure 8 uses the $\mathit{relevant}$ relation to express this; the first rule says that a bytecode is relevant if it is part of a $\mathit{derivedStream}$ relation that has been proven relevant, and the final three project any bytecode involved in main into the set of relevant bytecodes.

5. EXPERIENCES WITH PQL

Many of the error patterns found in the literature can be expressed easily in PQL. We have selected four important and representative error patterns to illustrate the use of PQL.

1. **Serialization errors**: a data corruption bug in web servers that can be exploited to mount denial-of-service attacks.

$$\begin{aligned}
&\mathit{main}(b_0, b_1, b_2, m_m, h_s, h_v, h_x, h_y) :- \\
&\qquad \mathit{IE}(c_0, b_0, \_, \text{“}\texttt{getInputStream}\text{”}), \\
&\qquad \mathit{actual}(b_0, 0, v_{s0}), \mathit{ret}(b_0, v_x), \\
&\qquad \mathit{vP}(c_0, v_x, h_x), \mathit{vP}(c_0, v_{s0}, h_s), \\
&\qquad \mathit{derivedStream}(\_, h_x, h_y, \_), \\
&\qquad \mathit{IE}(c_1, b_1,, \text{“}\texttt{readObject}\text{”}), \\
&\qquad \mathit{actual}(b_1, 0, v_{y1}), \mathit{ret}(b_1, v_{v1}), \\
&\qquad \mathit{vP}(c_1, v_{v1}, h_v), \mathit{vP}(c_1, v_{y1}, h_y), \\
&\qquad \mathit{IE}(c_2, b_2, \_, m_m), \mathit{actual}(b_2, 0, v_{v2}), \\
&\qquad \mathit{vP}(c_2, v_{v2}, h_v).
\end{aligned}$$

$$\begin{aligned}
&\mathit{derivedStream}(b, h_x, h_d, \_) :- \\
&\qquad h_x = h_d, b = \text{NULL}.
\end{aligned}$$

$$\begin{aligned}
&\mathit{derivedStream}(b, h_x, h_d, h_t) :- \\
&\qquad \mathit{IE}(c, b, \_, \text{“}\texttt{InputStream}. < \texttt{init} > \text{”}), \\
&\qquad \mathit{actual}(b, 1, v_x), \mathit{ret}(b, v_t), \\
&\qquad \mathit{vP}(c, v_x, h_x), \mathit{vP}(c, v_t, h_t), \\
&\qquad \mathit{derivedStream}(\_, h_t, h_d, \_).
\end{aligned}$$

$$\begin{aligned}
&\mathit{mainRelevant}(m_m, h_s, h_v, h_x, h_y) :- \\
&\qquad \mathit{main}(\_, \_, \_, m_m, h_s, h_v, h_x, h_y).
\end{aligned}$$

$$\begin{aligned}
&\mathit{derivedRelevant}(h_t, h_d, h_3) :- \\
&\qquad \mathit{derivedRelevant}(\_, h_d, h_t), \\
&\qquad \mathit{derivedStream}(\_, h_t, h_d, h_3).
\end{aligned}$$

$$\begin{aligned}
&\mathit{derivedRelevant}(h_x, h_y, h_3) :- \\
&\qquad \mathit{mainRelevant}(\_, \_, \_, h_x, h_y), \\
&\qquad \mathit{derivedStream}(\_, h_x, h_y, h_3).
\end{aligned}$$

$$\begin{aligned}
&\mathit{relevant}(b) :- \mathit{derivedStream}(b, h_x, h_d, h_t), \\
&\qquad \mathit{derivedRelevant}(h_x, h_d, h_t).
\end{aligned}$$

$$\mathit{relevant}(b) :- \mathit{main}(\_, \_, b, \_, \_, \_, \_, \_).$$

$$\mathit{relevant}(b) :- \mathit{main}(\_, b, \_, \_, \_, \_, \_, \_).$$

$$\mathit{relevant}(b) :- \mathit{main}(b, \_, \_, \_, \_, \_, \_, \_).$$

Figure 8: Datalog translation of Figure 5.

These errors are instances of the simple pattern “do not store object of type X in Y” [45].

2. **SQL injections**: a major threat to the security of database servers, as discussed in Section 1. This is an instance of “taint” analysis where the use of data obtained in some manner is restricted.

3. **Mismatched method pairs**: some APIs require that methods be invoked in a certain order. Matching pairs of methods that follow the pattern “a call to method A must *always* be followed by a call to method B” such as `install` *always* followed by `uninstall` are common in large systems. Failing to properly match method calls leads to resource leaks and data structure inconsistencies. Patterns of this kind are simple to specify, but are often difficult to check statically in large applications.

4. **Lapsed listeners**: a common memory leakage pattern in Java that may lead to resource exhaustion and crashes in long-running applications. Listeners follow a more complex pattern where event A invoked on an object is required to be followed by event B invoked on a related, but different object.

These examples considered together show the complementary nature of static and dynamic analysis. Static analysis can solve simple problems like the serialization error query precisely, whereas dynamic analysis becomes more useful for more complex queries like matched method pairs and lapsed listeners.

5.1 Experimental Setup

For our experiments, we use several large open-source Java applications whose characteristics are summarized in Figure 9. `webgoat` is a test application designed to demonstrate potential security flaws in Java. `road2hibernate` is a test program that exercises the rather large `hibernate` object persistence library, which is now a major component of the JBoss suite. `snipsnap`, `roller`, and `personalblog` are widely deployed weblog and wiki applications. Eclipse is the current premier open-source Java IDE; all Eclipse experiments in this paper were run on version 3.0.0.

All of our static analyses were done on an AMD Opteron 150 machine with 4GB of memory running Linux. Dynamic tests were performed on a 2 GHz AMD Athlon XP with 256MB of memory running Linux. First, we apply the context-sensitive pointer analysis on our benchmarks. As shown in Figure 10, it takes up to 34 minutes to represent the program as BDD relations and compute the points-to results. Fortunately, this preprocessing step only needs to be performed once for all queries. Note that even though `road2hibernate` consists of only 137 lines of code, the preprocessing time is dominated by the analysis of large libraries it uses.

In Figure 13 we show characteristics of the checkers for the three queries in our experiment. For static analysis, we show the time taken just to resolve the Datalog query and the total time taken, which is often considerably higher, as it includes loading and saving of large relations. Dynamic analysis is run only if warnings from the static analysis are not immediately obvious as errors. The times for the Web applications reflect the average amount of time required to serve a single page, as measured by the standard profiling tool JMeter. `road2hibernate` is a command-line program and its time is a simple start-to-finish timing.

Our performance numbers indicate that our approach on real applications is quite efficient. Unoptimized dynamic overhead is generally noticeable, but not crippling; after optimization it often becomes no longer measurable, though may still be as high as 37% in heavily instrumented code. Likewise, our static analysis times are in line with expectations for a context-sensitive pointer analysis run over tens of thousands of classes.

5.2 Serialization Errors

In a three-year study of production software, Reimer et al. found that a large class of high-impact coding errors violate design rules of the form “only store objects of type X in objects of type Y” [45]. Such rules can be easily expressed in PQL. The serialization error we study is an instance of such a pattern. Specifically, `HttpSession`, a runtime representation of a Web session, is supposed to be a persistent object to allow the Web server to save and restore sessions when the load is too high. As a consequence, only objects implementing the interface `Serializable` can be stored within an `HttpSession`, via the `setAttribute` method. The PQL query corresponding to this design rule is shown in Figure 11.

Violations of this rule will cause the persistence operation to fail, either with exceptions or via data corruption. The former may be exploited by a malicious user to mount a denial-of-service attack; the latter may cause intermittent problems that are hard to test because session objects are written out only under high load. One such problem in enterprise Java code reportedly took a team of engineers close to two weeks to detect [45].

Benchmark	Description	Source LOC	Source classes	Library classes	Total classes
webgoat	Sample Web application with known security flaws	19,440	35	986	1,021
personalblog	Blogging application based on J2EE	5,591	59	5,177	5,236
road2hibernate	Test application for Hibernate, an object persistence library	138	2	7,060	7,062
snipsnap	Blogging application based on J2EE	57,350	804	10,047	10,851
roller	Blogging application based on J2EE	52,089	247	16,112	16,359
Eclipse	Open-source Java IDE (GUI application)	2,834,133	19,439	—	19,439

Figure 9: Summary of information about benchmark Java programs.

Benchmark	Program relation generation	Pointer analysis	Total time
webgoat	65	13	78
personalblog	213	218	431
road2hibernate	767	512	1,279
snipsnap	170	151	321
roller	978	1,011	2,029

Figure 10: Static preprocessing time, in seconds.

Benchmark	Total calls	Static warnings	Stat. confirmed errors	Dynam. confirmed errors
webgoat	5	1	1	—
personalblog	2	0	0	—
snipsnap	29	10	6	1
roller	25	1	1	—
Total	61	12	8	1

Figure 12: Results for the serialization error query. Calls refer to invocations of `HttpSession.setAttibute`. "—" indicates that dynamic checking is unnecessary.

As shown in Figure 12, a total of 61 calls to method `HttpSession.setAttribute` are found in four benchmarks. After the optimizer was run, only 12 remain as potential matches to our query. This shows how pointer analysis is useful in suppressing false warnings: the static checker is able to deduce that the concrete types of the instances stored implement `Serializable` in some cases, even though their declared type is not. 8 of the remaining calls to `setAttribute` are obvious errors that can immediately be seen to not be correct on any run. When our dynamic checker is applied to `snipsnap`, which contains the 4 unconfirmed warnings, a runtime match is found for one of these suspicious sites, confirming that it is indeed an error.

5.3 Finding Security Flaws: SECURIFLY

Shown in Figure 14 is a more realistic example of the SQL injection vulnerability first mentioned in Section 1.1. Having control over the `username` and `pwd` variables, the user can cause arbitrary SQL code to be run or bypass access restrictions. SQL injection is an instance of "taint" analysis which requires tracking the flow of data from a set of *sources* to a set of *sinks*.

For applications written in the J2EE framework, we have examined the J2EE APIs to identify the sources and sinks for the case of SQL injections. Sources, listed in query `UserSource` in Figure 15, include return results of `HttpServletRequest`'s methods such as `getParameter`. Sinks, enumerated in the `replaces` clause, include arguments of method `java.sql.Statement.execute(String sql)`, `java.sql.Connection.prepareStatement(String sql)`, and so forth.

Because a user-controlled string may be incorporated into other strings, the main query asks if a user-controlled string (subquery `UserSource`), can be propagated one or more times (subquery `StringPropStar`) to create a string used in an SQL query (the actions in the `replaces` clauses of the main query). Unsafe database accesses are replaced with routines that first quote every metacharacter in every instance of the user string in the SQL command, thus transforming possible attacks into legitimate commands.

Note that the string propagation query `StringPropStar` is not specific to SQL injection, and can be used for a variety of taint queries that involve propagation of `Strings`. It invokes the `StringProp` query, which handles all the ways in which one string can be derived from another.

Using PQL we have developed a runtime security protection system for Web applications called SECURIFLY[1]. The system presented here can address the problem of SQL injection as well as other vulnerabilities such as cross-site scripting and path traversal attacks described in [33]. However, we have only performed a detailed experimental study of runtime overhead for SQL injections.

Commonly used dynamic techniques such as application firewalls [37] that rely on pattern-matching and monitor traffic flowing in and out of the application are a poor solution for SQL injection [35]. In contrast, SECURIFLY can detect attacks because it observes how data flows through the application. Moreover, SECURIFLY can gracefully recover from vulnerabilities before they can do any harm by sanitizing tainted input whenever necessary. There are some inherent advantages the dynamic approach has over the static one.

- SECURIFLY can be integrated with the server so that whenever a new Web application is added, it is instrumented automatically. This removes the apprehension related to deploying "unfamiliar" potentially insecure Web applications. This obviates the issue present with static tools of the code being changed without the tool being rerun. This is particularly important because analyzing Web applications statically can prove to be difficult because of issues such as handling reflection.

```
query main()
returns
    object !java.io.Serializable obj;
    object javax.servlet.http.HttpSession session;
matches {
    session.setAttribute(_, obj);
}
```

Figure 11: Query for finding serialization errors.

[1]The name SECURIFLY comes from the idea of "providing security on the fly."

	Static analysis time		Instrumentation points		Runtime			Overhead	
Benchmark	Query resolution	Total time	Unopti-mized	Opti-mized	Uninstru-mented	Unopti-mized	Opti-mized	Unopti-mized	Optimi-mized
	BAD STORES								
`webgoat`	5	12	1	0	—	—	—	—	—
`personalblog`	23	34	1	0	—	—	—	—	—
`snipsnap`	48	67	18	3	.073	.074	.073	1%	$< 1\%$
`roller`	61	84	12	0	—	—	—	—	—
	SQL INJECTIONS								
`webgoat`	1	46	604	69	.024	.054	.033	125%	37%
`personalblog`	2	74	3,209	36	.040	.069	.049	72%	22%
`road2hibernate`	4	113	4,146	779	2.224	2.443	2.362	9%	3%
`snipsnap`	3	79	3,305	542	.073	.096	.080	31%	9%
`roller`	4	147	2,960	96	.008	.012	.008	50%	$< 1\%$

Figure 13: Summary of static analysis times, runtimes, dynamic overhead, and the number of instrumentation points with and without optimizations. "—" is used to indicate that no dynamic run was necessary because a static solution was sufficient. All times are in seconds.

- SECURIFLY does not require changes to the original program and does not need access to anything other than the final bytecode. This can be especially advantageous when dealing with applications that rely on libraries whose source is unavailable.

The dynamic checker for the SQL injection query will match whenever a user controlled string flows in some way to a suspected sink, regardless of whether a user input is harmful in a particular execution. It will then react to replace the potentially dangerous string with a safe one.

The errors located with our tool involved the applications building SQL strings out of data either sent in from the command line or generated as parameters of an HTTP request. The former can be exploited if the program can be executed by the malicious user. The latter are vulnerable to the more common crafted-HTTP-request attacks.

5.3.1 *Importance of Static Optimization*

Without static optimization, many program locations need to be instrumented. This is because routines that cause one `String` to be derived from another are very common. Heavily processed user inputs that do not ever reach the database will also be carefully tracked at runtime, introducing significant overhead to the analysis.

Fortunately, the static optimizer effectively removes instrumentation on calls to string processing routines that are not on a path from user input to database access. Exploiting pointer information dramatically reduces both the number of instrumentation points and the overhead of the system, as shown in Figure 13. The reduction in the number of instrumentation points due to static optimization can be as high as 97% in `roller` and 99% in `personalblog`. As shown in Figure 13, reduction in the number of instrumentation points results in a smaller overhead. For instance, in `webgoat`, the overhead is cut almost in half in the optimized version.

```
public void authenticate(HttpServletRequest request){
    String username = request.getParameter("user");
    java.sql.Statement stmt = con.createStatement();

    String query =
        "select * from users where username = '" +
        username + "'and password = '" + pwd + "'";

    stmt.execute(query);
    ... // process the result of SELECT
}
```

Figure 14: A classic example of SQL injection.

Note that the query does no direct checking of the value that has been provided by the user, so if harmless data is passed along a feasible injection vector, it will still trigger a match to the query. As a result of this, drastic responses such as aborting the application are not suitable outside of a debugging context.

5.3.2 **Applying Input Sanitization**

As seen in Figure 15, each operation that can unsafely use tainted data receives a `replaces` clauses in the query `main`. When a possibly relevant sink is reached, any matches that have completed and which are consistent with the instruction are gathered, and if such matches are present, the replacing method is executed instead.

The `SafePrepare` and `SafeExecute` methods themselves find all substrings in the `sink` variable that match any of the possible values for `source`. They then produce a new SQL Query string identical to the old, but it quotes all the SQL metacharacters such as "'". This forces them to be treated as literal characters instead of, for instance, a string terminator. This new, safe query is then passed to `prepareStatement` or `executeQuery`, respectively.

Using this technique we were able to defend against the two SQL injections for which we had derived effective attacks: the two in `webgoat` and `road2hibernate`.

5.4 Matching Method Pairs in Eclipse

Many APIs have methods that must be invoked in pairs in order for to remain internally consistent or to prevent resource leaks. This implies a set of rules that take the form "a call to A must *always* be followed by a call to B." Developers are required to ensure that calls to A are followed by calls to B on all execution paths—a highly error-prone task, especially in the presence of exceptions. The problem is complicated further when calls to A and B occur in different classes, or when a subclass overrides the method that calls B but not the one that calls A. As a result, not only is the process error-prone, it is difficult to debug with traditional techniques.

Eclipse, a large open-source Java IDE, uses a windowing toolkit called SWT [41], which has many examples of such method pairs. Programming guides and bug reports directed us to eight examples of paired initialize/uninitialize methods that were often violated. For example, calls to `createWidget` must always be followed to a call to `destroyWidget` on the same object. We instrumented Eclipse to search for instances of the first with no corresponding call to the second. This is done with a query body of the form

$$\{\texttt{o.A}(); \sim \texttt{o.B}(); \}$$

As these are liveness queries that rely critically on excluding the second event, and our static analysis is flow-insensitive, we cannot

```
query main()
returns
    object Object source, sink;
uses
    object java.sql.Connection con;
    object java.sql.Statement stmt;
matches {
    source := UserSource();
    sink   := StringPropStar(source);
} replaces con.prepareStatement(sink)
        with SQL.SafePrepare(con, source, sink);
   replaces stmt.executeQuery(sink)
        with SQL.SafeExecute(stmt, source, sink);


query StringProp(object Object x)
returns
    object Object y;
matches
      y.append(x)
    | y = new String(x)
    | y = new StringBuffer(x)
    | y = x.toString()
    | ...


query StringPropStar(object Object x)
returns
    object Object y;
uses
    object Object temp;
matches
    y := x |
    {
        temp  := StringProp(x);
        y := StringPropStar(temp);
    }


query UserSource()
returns
    object Object tainted;
uses
    object ServletRequest req;
matches
    tainted = req.getParameter() |
  | tainted = req.getHeader() | ...
```

Figure 15: Full SQL injection query.

optimize the dynamic matcher. However, even without optimization, there are still relatively few instrumentation points because the interesting events are confined to the creation and destruction of GUI elements. Our approach is powerful because this one-line query can pick the handful of points out of nearly 64 MB of classfiles that actually need to be instrumented for the query directly.

The results of running the instrumented IDE for each method pair are summarized in Figure 16, together with the number of instrumentation points and dynamic violations of each patterns. The number of dynamically discovered errors reported in the table is the number of concrete types of object *o* that violated the pattern. A total of 56 types in Eclipse had pattern violations, most of which were due to missing calls to `destroyWidget`. In extended runs, these bugs will lead to resource leaks.

5.5 Memory Leaks: Lapsed Listeners

Paired method queries, while useful, do not require many of PQL's features to perform. To demonstrate the ability of PQL to correlate objects in widely spaced events, we formed a query to discover memory leaks in Eclipse. Despite being garbage collected, Java programs can still have memory leaks [51]. A Java program can maintain a link to an object that is never used again, causing the garbage collector to never reclaim that object. Finding such kinds of memory leaks is difficult, but it is important to find them because they can gradually cause resource exhaustion in long-running applications such as Web servers, leading to instability and crashes.

Event listeners in Java GUI programs are a common source of memory leaks. Event listeners are a common way to specify actions that should occur when a user interface event such as a mouse click occurs on a given GUI component. This is achieved by *registering* a listener with a GUI component; when the component is destroyed, the listener should be *unregistered*. If a listener is not unregistered, it will preserve a link to the GUI component. In Swing and SWT, the listener is reachable from a global listener table, thus making the GUI component also reachable and therefore considered live by the garbage collector. This is referred to in the literature as the *lapsed listener problem* [51].

5.5.1 Listeners in Eclipse

Eclipse's GUI library is vulnerable to the lapsed listener problem. Some lapsed listener errors have been featured on Eclipse's bug tracking system. The usual technique for finding these errors is to spend a lot of time inspecting code and using heap debuggers.

The API in Eclipse works as follows. Components (`ViewParts`) are created and destroyed with the `createPartControl` and `dispose` methods. Listeners must be registered within `createPartControl`, and unregistered within `dispose`. A query for this is given in Figure 17. (The actual query has many possibilities for the registration and deregistration of listeners; for simplicity our example uses one representative method for each.)

Like paired methods, this query relies heavily on the use of excluded events, which the static analysis currently ignores. As a result, the static analysis will simply report the presence of `createPartControl` calls that register listeners, and are then `disposed`, which is not useful. We thus again apply only unoptimized dynamic analysis to Eclipse. Overhead was not perceptible during runs of the instrumented application.

The results of this are shown in the last line of Figure 18. Matches were collected by type. After performing work in the Java, Resource, and CVS perspectives, 136 `ViewPart`/`Component`/`Listener` type triples were found that were not disposed of according to the model.

This experiment demonstrated that PQL queries can be used effectively even on extremely large programs, and to find properties that resist most forms of analysis.

5.6 Summary

In this section we have described four applications representing a spectrum as far as being amenable to static and dynamic analysis. Serialization errors can be addressed statically with a high degree of precision using high-quality pointer information. SQL injection, while benefitting from static analysis to reduce the amount of instrumentation, cannot generally be fully and precisely addressed using our static technique, but is a good match for dynamic monitoring optimized with the help of static results. Finally, liveness properties such as paired methods and lapsed listener problems found in `Eclipse` lend themselves most naturally to dynamic analysis. Furthermore, instrumented executables produced by PQL provide protection against query violations at run time. We have discovered a total of 9 serialization errors, 5 SQL injections, and nearly 200 errors in Eclipse, all of which were previously unknown.

Query $\{$o.A; $\sim$ o.B$\}$ Method A	Method B	Instrumentation points	Dynamically discovered errors
`createWidget`	`destroyWidget`	28	35
`register`	`deregister`	93	7
`acquireXMLParsing`	`releaseXMLParsing`	2	0
`acquireDocument`	`releaseDocument`	12	0
`install`	`uninstall`	184	10
`installBundle`	`uninstallBundle`	11	0
`start`	`stop`	120	4
`startup`	`shutdown`	41	0
Total		491	56

Figure 16: Result summary for running matching pair queries on Eclipse.

```
query creation(object ViewPart vp)
returns
    object Object obj;
    object Object listener;
within vp.createPartControl() matches {
    obj.registerListener(listener);
}

query destroy(object ViewPart vp, object Object obj,
                         object Object listener)
within vp.dispose() matches {
    ∼obj.unregisterListener(listener);
}

query main()
returns
    object ViewPart    vp;
    object Object      obj;
    object Object      listener;
matches {
    (obj, listener) := creation(vp);
    ∼obj.unregisterListener(listener);
    () := destroy(vp, obj, listener);
}
```

Figure 17: Lapsed listener query.

6. APPLICATIONS OF PQL

The topics covered in Section 5 focused narrowly on specific types of errors. In this section, we move to a higher vantage point and discuss more general uses of PQL.

6.1 Debugging and Testing

We primarily envision PQL being used under development or debugging conditions. When a developer finds a bug in their code, they will often suspect that similar bugs lurk elsewhere in their code base. Over time, the collection of such rules that encodes developers' knowledge of system-specific rules grows. The essence of many bug patterns, from simple mistakes that manifest on single lines (like the serialization errors in Section 5.2, or ensuring that private data members cannot be modified because references to them escape [22]) to ordering constraints across the life of the entire program (as demonstrated in Sections 5.4 and 5.5), is often directly expressible as a simple PQL query. However, in large projects with multiple authors, finding all similar instances of the error is not a trivial task, and one with which our system can help considerably.

Instrumentation Points	Dynamically discovered errors
13,661	136

Figure 18: Result summary for lapsed listeners in Eclipse.

In general, *automatically* discovering system-specific rules is a challenging problem well-represented in the literature [16, 57, 32]. Below we outline some queries gleaned from the literature and our own programming experience.

6.1.1 Taint Queries

The SQL injection problem, discussed in Section 5.3 is an instance of a general class of *tainted data problems*, where data originates at a source, is propagated via various techniques, and then used as a sink. Many other practical security problems fall into this class. In the Web application domain, cross-site scripting [50] and HTTP splitting [26] attacks are two more techniques that use unsanitized data in order to mount the attack. Below we discuss several other applications of taint queries for security.

Unsafe manipulation of password strings. It is not uncommon to have methods in Java APIs that take `char`[ ] arrays representing passwords as a parameter. Several such functions from publicly available APIs are listed in Table 19. Character arrays are recommended over `String`s as a way to reduce the window of vulnerability; character arrays are zeroed-out after the function returns. This way, the duration of time when the password is in memory is minimized, reducing the window of opportunity for a hacker combing through memory looking for clear-text passwords [34].

However, in practice many developers using these APIs are unaware of this security hazard and construct password strings by calling `String.getChars`(). The character array representing internals of the `String` remains in memory after the return result of `getChars`() has been zeroed-out, thus defeating the point of passing in a `char`[ ] in the first place. This security issue is another instance of the tainted data problem: the set of sources for this taint problem is the return results of `String.getChars`(); a sample set of sinks is the password parameters of functions in Table 19.

Leaking sensitive data. Data such as user login and password or financial information like credit card numbers need to be protected from unauthorized access. If this data is saved in cookies or application logs without being properly encrypted, eavesdroppers may be able to gain access to this potentially sensitive data. Therefore, *output sanitization* that removes sensitive data is required before saving.

Another common error pattern that involves reporting sensitive data back to the user by showing extra information in an exception trace [56]. It is therefore crucial to find how data obtained from databases propagates to output functions pertaining to file or socket output streams or API-specific functions such as J2EE routines for manipulating cookie data. This example too is expressed as a taint problem similar to the queries used in Section 5.3.

`javax.crypto.spec.PBEKeySpec(char[ ] password)`
`gnu.crypto.keyring.PasswordAuthenticatedEntry.authenticate(char[ ] password)`
`edu.uidaho.hummer.util.DefaultKeyStore.loadData(char[ ] password)`
`com.mindbright.security.keystore.PKCS12KeyStore.deriveKey(char[ ] password, ...)`

Figure 19: Public API functions taking `char[ ]` password parameters.

6.1.2 Fault Injection

So far, all the event replacement situations we have discussed involve replacing unsafe arguments with safe ones to ensure security. One could also deliberately attempt to insert *unsafe* values to test the robustness of a system. Fault injection is a testing technique that "injects" unexpected data into the application in order to cause logical or security errors and crashes [14, 15, 53, 61]. PQL's event replacement functionality allows the user to easily describe fault injection rules and create fully functional testing frameworks.

For example, Java applications may be compromised by passing potentially unexpected parameters to `native` Java calls. There are hundreds of `native` calls in public Java APIs that take `Object` or array parameters. All these methods are typically implemented in C or C++ on the platform to which the APIs are ported, thus creating a potential for buffer overrun attacks in the native code. In the past, fault injection attacks that pass very large arrays or `null` objects to `native` methods have been used to successfully compromise JDK implementations from multiple vendors [48]. These attacks often result in the Java Virtual Machine crashing, and can also result in obtaining root privileges if a hacker manages to craft an appropriate array parameter.

6.2 Program Exploration

Developers beginning work on a large existing project usually have a daunting task ahead of them. Determining how pieces of a large system fit together through code inspection is difficult and time-consuming. PQL can be used to extract information about how objects are actually used as the program runs. We have ourselves used this approach when developing and testing PQL, to determine if perceived program invariants actually held reliably.

6.2.1 Finding Application Entry Points

Modern middleware systems are designed to support multiple user-provided components; these include *plugins* in Eclipse, *servlets* in J2EE, and *modules* in Apache. In many cases, the underlying structure is complex and it is often unclear in what way the user-provided code gets called by the system. Analyzing such modules statically, however, generally requires information on application entry points to prime the analysis.

There are two strategies one can follow to find this information dynamically in PQL. One is to instrument the framework and look for calls where the `this` pointer is of a type defined by the target module, and log such information. The other is to trap calls *within* the application and then examine the stack trace for where control transfers from the framework to the application.

6.2.2 Discovering Data Validation Strategies

Most Web-based applications use some sort of a validation scheme to prevent user-provided data from being used in the program in unsafe ways as described in Section 5.3. Typically, user-provided data is checked using regular expressions or encoded using URL-encoding or similar schemes. Furthermore, sometimes data is stored or passed around in an encrypted format. PQL can help us discover what perturbations occur to data after it has entered the application. Our preliminary experiments reveal that in many cases URL-encoding and regular expression pattern matching is applied on user-provided data as a means of cleansing it.

6.2.3 Finding Instantiated Types

Java libraries frequently export only abstract types to the application layer so that varying library implementations may be connected to the same class. It is, by design, unclear precisely which classes are actually being used in such an application. This complicates many interprocedural static analyses, because they cannot readily find a useful call graph.

This problem is particularly pernicious in the presence of reflection, because call graph information is often contained in configuration files instead of the code itself. It can be convenient to extract from runs of the program precisely which classes are being loaded reflectively, and use this to construct the analysis's call graph. In Java, the method `Class.forName(String className)` is used to get an object of type `Class` that represents the type named in `className`, and then the `newInstance()` method actually constructs the object. The result of this is typically then cast to the an abstract class or interface that the application then uses thereafter.

If one wishes to focus on instances of a specific supertype, the query may be focused easily. For example, in the process of exploring the SQL injection issue in Section 5.3, it was necessary to find which subclasses of `java.sql.Statement` were instantiated for a given program.

7. RELATED WORK

7.1 Model Extraction

Some work has been done on automatically inferring state models on components of software systems. The Strauss system [4] uses machine learning techniques to infer a state machine representing the proper sequence of function calls in an interface. Redux [38] studies a program as it runs and builds up a tree of value dependencies that capture the "essence" of the computation. Whaley et al. [60] hardcode a restricted model paradigm so that probable models of object-oriented interfaces can be easily automatically extracted. Alur et al. [3] generalize this to automatically produce small, expressive finite state machines with respect to certain predicates over an object. Lam et al. use a type system-based approach to statically extract interfaces [28]. Their work is more concerned with high-level system structure rather than low-level life-cycle constraints [47]. Daikon is able to validate correlations between values at runtime and is therefore able to validate patterns [16]. Weimer et al. use exceptional control-flow paths to guide the discovery of temporal error patterns with considerable success [57]; they also provide a comparison with other existing specification mining work. DynaMine uses CVS history mining combined with dynamic analysis to discover good patterns and to detect pattern violations [32].

In contrast to these, the PQL system places the burden of model generation on the user. However, partial models may be used to develop queries that provide information that suggests how to extend the model. With suitable actions attached to the queries, a PQL query can be used to implement a specialized model extractor directly.

7.2 Aspect-Oriented Programming

PQL attaches user-specified actions to subquery matches; this capability puts PQL in the class of *aspect-oriented* programming languages [25, 43]. Maya [7] and AspectJ [25] attach actions based on syntactic properties of individual statements in the source code. The DJ system [43] defines aspects as traversals over a graph representing the program structure. Our system may be considered as an aspect-oriented system that defines its aspects with respect to the dynamic history of sets of objects. An extension of AspectJ to include "dataflow pointcuts" [36] has been proposed to represent a statement that receives a value from a specific source; PQL can represent these with a two-statement query, and permits much more complex concepts of data flow. Walker and Veggers [55] introduce the concept of *declarative event patterns*, in which regular expressions of traditional pointcuts are used to specify when advice should run. Allan et al. [2] extend this further by permitting PQL-like free variables in the patterns.

The primary focus in the AspectJ extensions is in permitting a developer to specify application development concerns very finely. As a result, they devote a great deal of work to ensuring properties such as guarantees that memory allocated by the matching machinery will eventually be available for collection. PQL, with its genesis focusing on detecting application errors, pays less attention to this. (For example, one of the patterns they warn against involves paired methods across objects, because such patterns are intrinsically leaky. This class, however, includes our lapsed listener example.)

PQL differs from these systems in that its matching machinery can recognize nonregular languages, and in exploiting advanced pointer analysis to prove points irrelevant to eventual matches.

7.3 Program Defect Detection

A vast amount of work has been done in bug detection. C and C++ code in particular is prone to buffer overrun and memory management errors; tools such as PREfix [10] and Clouseau [20] are representative examples of systems designed to find specific classes of bugs (pointer errors and object ownership violations respectively). Dynamic systems include Purify [19], which traps heap errors, and Eraser [46], which detects race conditions. Both of these analyses have been implemented as standard uses of the Valgrind system [39]. Many of these bug classes are outside the purview of the PQL language at present, focusing as they do on the use of individual pointer variables or on synchronization primitives. PQL also targets Java, which means

Web applications carry their own set of security risks [44]. Various systems have been developed to help secure web applications. SABER [45] is a static tool that detects a large number of common design errors based on instantiations of a number of error pattern templates. WebSSARI [23] and Nguyen-Tuong et al. [40] are dynamic systems that detects failures to validate input and output in PHP applications. While PQL does not handle PHP, in principle these analyses perform sequencing, type, or tainting analysis and as such are easily amenable to representation as PQL queries directly. The latter project is suitable for tracking taintedness at a much finer granularity. configurable to express alternate patterns.

7.4 Event-based Analysis

The queries in our system are defined with respect to a conceptual abstract execution trace consisting of a stream of events. The implications of this paradigm for debugging are covered extensively in the EBBA system [9]; later tools have expanded on the basic concept to provide additional power. Dalek [42] is a debugger that defines compound events out of simpler ones, and permits breakpoints to occur only when a compound event has executed. PQL follows Dalek in building its queries out of patterns of simple events, and builds upon it by permitting the events to be recursively (and, indeed, even mutually recursively) defined.

7.5 Other Program Query Languages

Systems like ASTLOG [13] and JQuery [24] permit patterns to be matched against source code; Liu et al. [31] extend this concept to include parametric pattern matching [6]. These systems, however, generally check only for source-level patterns and cannot match against widely-spaced events. A key contribution of PQL is a pattern matcher that combines object-based parametric matching across widely-spaced events.

Lencevicius et al. developed an interactive debugger based on queries over the heap structure [29]. This analysis approach is orthogonal both to the previous systems named in this section as well as to PQL; however, like PQL, its query language is explicitly designed to resemble code in the language being debugged.

7.5.1 Partiqle

The Partiqle system [17] uses a SQL-like syntax to extract individual elements of an execution stream. It does not directly combine complex events out of smaller ones, instead placing boolean constraints between primitive events to select them as sets directly. Variables of primitive types are handled easily by this paradigm, and nearly arbitrary constraints can be placed on them easily, but strict ordering constraints require many clauses to express.

This reliance on individual predicates makes their language easy to extend with unusual primitives; in particular, the Partiqle system is capable of trapping events characterized by the amount of absolute time that has passed, a capability not present in the other systems discussed. However, like most other systems, it can still only quantify over a finite number of variables. PQL's recursive subquery mechanism makes it possible to specify arbitrarily long chains of data relations.

For comparison purposes, we reproduced their `StringConcats` experiment in PQL, which is the only query they performed that operates solely on objects. The PQL query itself is a single straight-line sequence of calls, chained together five times; we applied the query to a number of the SPECJVM98 benchmarks, including all of the four that do extensive string processing. The overhead, expressed as a ratio between instrumented and uninstrumented runtime, is given in Figure 20.

The worst-case scenario for the PQL matcher is one in which many instances of the beginning of a pattern appear, thus producing a large number of partial matches to track. This query begins matching any time a `StringBuffer` is created, which is a very common operation. In the highest-overhead case here—the `jack` benchmark—a complete match occurs for every execution of the inner loop of the code reading the input file. This loop reads a single character each iteration, producing thousands of matches over the course of the run.

In cases where string use is clear, amenable to static pointer analysis, and does not contribute to matches—such as `jess` and `compress`—static optimization is able to lessen overhead dramatically, often to the point that overhead is lost in the instrumentation noise.

In programs where string processing is simply rare, such as `db`, no measurable cost is ever imposed.

7.6 Analysis Generators

PQL follows in a tradition of powerful tools that take small specifications and use them to automatically generate analyses.

Benchmark	`mtrt`	`jess`	`compress`	`db`	`jack`	`javac`
Unoptimized	1.1	2.3	1.1	1.0	19	2.7
Optimized	1.0	1.2	1.0	1.0	19	2.7

Figure 20: Overhead ratios for the string concatenation query on SPECJVM.

Metal [18] and SLIC [8] both define state machines with respect to variables. These machines are used to configure a static analysis that searches the program for situations where error transitions can occur. Metal restricts itself to finite state machines, but has more flexible event definitions and can handle pointers (albeit in an unsound manner).

The Rhodium language [30] uses definitions of dataflow facts combined with temporal logic operators to permit the definition of analyses whose correctness may be readily automatically verified. As such, its focus is significantly different from the other systems, as its intent is to make it easier to directly implement correct compiler passes than to determine properties of or find bugs in existing applications. Likewise, though it is primarily intended as a vehicle for predefined analyses, Valgrind [39] also presents a general technique for dynamic analyses on binaries.

7.7 Model Checkers

Model checking systems such as SPIN [21] are powerful and widespread tools for capturing complicated program properties. Model checkers generally operate upon abstract languages such as Promela; the Bandera project [11] abstracts Java code into a form amenable to SPIN and other model checkers. These systems represent queries over the models as LTL formulas on predicates—Bandera ties these predicates to expressions defined in the code itself [12].

This paradigm does not lend itself well to direct comparison with PQL, as the predicates implied by a PQL query relate to object identity across statements and to the histories of those objects or sets of objects. It is possible to consider any individual PQL query as a temporal logic equation describing the events themselves, existentially quantified over objects on the heap. Arbitrary LTL equations on such a set of predicates would not be directly translatable into PQL. Also, LTL-based model checkers quantify universally over paths (ensure the specified formula holds on all paths), while PQL quantifies existentially over paths and over the heap (find a set of objects such that the pattern is matched over some path). PQL's subquery mechanism permits already-determined matches to be treated as atomic propositions for higher-level (or even recursive) queries, which makes expressing many properties much easier.

Further research by the Bandera group into the utility of LTL for program analysis has resulted in a catalogue of *specification patterns* [49]—idioms in LTL that are particularly prevalent in useful queries—and these idioms correspond to presence, absence, or sequencing of individual events. These patterns are directly representable in PQL via the use of nothing more than the sequencing operator and match exclusion.

Thus, PQL's language can be viewed as complementary to LTL-based systems: both are capable of expressing the "natural" practical queries for application programs, but the underlying concepts are different (LTL's concept of predicates changing truth values over time compared to PQL's concept of objects evolving through various states with time) and each system thus generalizes into different sets of applications.

8. CONCLUSIONS

We have presented a new language called PQL, with which users can pose application-specific questions about events patterns at runtime. The language is intuitive to use by application developers and provides a bridge to powerful analysis techniques.

We systematically convert PQL queries into efficient online checkers by using a combination of static and dynamic techniques. In so doing, we demonstrate an application for pointer analysis in bug-finding in which the soundness of the analysis is critical.

Using PQL, we have found numerous previously unknown security vulnerabilities and resource leaks in large open-source Java applications. Our experience suggests that the synergistic combination of static and dynamic checking is a powerful one. Static analysis can find all potential errors in some of the cases; in others, it can prove that the pattern will never match. In more difficult cases, dynamic monitoring can guarantee that applications can trap any instance of a certain class of errors. Static analysis is also useful here for reducing dynamic analysis overhead. With static optimizations, we have found dynamic checking to often have a low enough overhead to be incorporated in production code.

9. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[2] C. Allan, P. Augustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA '05: Proceedings of the 20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[3] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java Classes. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 98–109, 2005.

[4] G. Ammons, R. Bodik, and J. Larus. Mining Specifications. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[5] C. Anley. Advanced SQL Injection in SQL Server Applications, 2002.

[6] B. S. Baker. Parameterized Pattern Matching by Boyer-Moore Type Algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms,* pages 541–550, 1995.

[7] J. Baker and W. C. Hsieh. Runtime Aspect Weaving Through Metaprogramming. In *Proceedings of the First International Conference on Aspect-Oriented Software Development,* 2002.

[8] T. Ball and S. Rajamani. SLIC: A Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, January 2002.

[9] P. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed Debugging,* pages 11–22, 1988.

[10] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software - Practice and Experience (SPE)*, 30:775–802, 2000.

[11] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting Finite-State Models from Java Source Code. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, 2000.

[12] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A Language Framework for Expressing Checkable Properties of Dynamic Software. In *SPIN '00: Proceedings of the 7th SPIN Workshop*, pages 205–223, 2000.

[13] R. F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 229–242, 1997.

[14] W. Du and A. P. Mathur. Vulnerability Testing of Software System Using Fault Injection. Technical report, COAST, Purdue University, West Lafayette, IN, US, April 1998.

[15] W. Du and A. P. Mathur. Testing for Software Vulnerability Using Environment Perturbation. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000), Workshop On Dependability Versus Malicious Faults*, pages 603–612, New York City, NY, June 2000.

[16] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly Detecting Relevant Program Invariants. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, Limerick, Ireland, June 7–9, 2000.

[17] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational Queries Over Program Traces. In *Proceedings of the ACM SIGPLAN 2005 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[18] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific, Static Analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 69–82, 2002.

[19] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, pages 125–136, December 1992.

[20] D. Heine and M. S. Lam. A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 168–181, 2003.

[21] G. J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[22] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. In *Proceedings of the Onward! Track of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.

[23] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the 13th Conference on the World Wide Web*, pages 40–52, 2004.

[24] D. Janzen and K. de Volder. Navigating and Querying Code Without Getting Lost. In *Proceedings of the 2nd Annual Conference on Aspect-Oriented Software Development (AOSD)*, pages 178–187, 2003.

[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[26] A. Klein. Divide and Conquer: HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics. `http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf`, 2004.

[27] S. Kost. An Introduction to SQL Injection Attacks for Oracle Developers, 2004.

[28] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, pages 275–302, Darmstadt, Germany, July 2003.

[29] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-Based Debugging of Object-Oriented Programs. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 304–317, New York, NY, USA, 1997. ACM Press.

[30] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated Soundness Proofs for Dataflow Analyses and Transformations Via Local Rules. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 364–377, 2005.

[31] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu. Parametric Regular Path Queries. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, pages 219–230, 2004.

[32] B. Livshits and T. Zimmermann. DynaMine: A Framework for Finding Common Bugs by Mining Software Revision Histories. In *Proceedings of the ACM SIGSOFT 2005 Symposium on the Foundations of Software Engineering (FSE)*, Sept. 2005.

[33] V. B. Livshits and M. S. Lam. Finding Security Errors in Java Programs with Static Analysis. In *Proceedings of the 14th Usenix Security Symposium*, Aug. 2005.

[34] Q. H. Mahmoud. Password Masking in the Java Programming Language. `http://java.sun.com/developer/technicalArticles/Security/pwordmask/`, July 2004.

[35] F.-M. S. mailing list. Vulnerability Scanner for SQL injection. `http://www.derkeiler.com/Mailing-Lists/securityfocus/focus-ms/2003-09/0110.html`, 2003.

[36] H. Masuhara and K. Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. In *APLAS'03 - the First Asian Symposium on Programming Languages and Systems*, pages 105–121, 2003.

[37] Netcontinuum, Inc. Web Application Firewall: How NetContinuum Stops the 21 Classes of Web Application Threats. `http://www.netcontinuum.com/products/whitePapers/getPDF.cfm?n=NC_WhitePaper_WebFirewall.pdf`, 2004.

[38] N. Nethercote and A. Mycroft. Redux: A Dynamic Dataflow Tracer. In O. Sokolsky and M. Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[39] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In O. Sokolsky and M. Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[40] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, 2005.

[41] S. Northover and M. Wilson. *SWT : The Standard Widget Toolkit, Volume 1*. Addison-Wesley Professional, 2004.

[42] R. A. Olsson, R. H. Cawford, and W. W. Ho. A Dataflow Approach to Event-Based Debugging. *Software - Practice and Experience*, 21(2):209–230, 1991.

[43] D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns* , Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[44] OWASP. Ten Most Critical Web Application Security Vulnerabilities, 2004.

[45] D. Reimer, E. Schonberg, K. Srinivas, H. Srinivasan, B. Alpern, R. D. Johnson, A. Kershenbaum, and L. Koved. SABER: Smart Analysis Based Error Reduction. In *Proceedings of International Symposium on Software Testing and Analysis*, 2004.

[46] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[47] S. R. Schach. *Object-Oriented and Classical Software Engineering*. McGraw-Hill Science/Engineering/Math, 2004.

[48] M. Schonefeld. Hunting Flaws in JDK. In *Blackhat Europe*, 2003.

[49] `http://patterns.projects.cis.ksu.edu/`.

[50] K. Spett. Cross-Site Scripting: Are Your Web Applications Vulnerable. `http://www.spidynamics.com/support/whitepapers/SPIcross-sitescripting.pdf`, 2002.

[51] B. A. Tate. *Bitter Java*. Manning Publications Co., 2002.

[52] M. Vernon. Top Five Threats. ComputerWeekly.com (http://www.computerweekly.com/Article129980.htm), April 2004.

[53] J. Voas and G. McGraw. *Software Fault Injection: Innoculating Programs Against Errors*. John Wiley and Sons, 1997.

[54] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 3–17, 2000.

[55] R. J. Walker and K. Viggers. Implementing Protocols Via Declarative Event Patterns. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 159–169, New York, NY, USA, 2004. ACM Press.

[56] Web Application Security Consortium. Threat Classification. `http://www.webappsec.org/tc/WASC-TC-v1_0.pdf`, 2004.

[57] W. Weimer and G. Necula. Mining Temporal Specifications for Error Detection. In *Proceedings of the 11th International Conference on Tools and Algorithms For The Construction And Analysis Of Systems*, pages 461–476, Apr. 2005.

[58] W. Weimer and G. C. Necula. Finding and Preventing Run-Time Error Handling Mistakes. In *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, Oct. 2004.

[59] 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 (PLDI)*, 2004.

[60] J. Whaley, M. Martin, and M. S. Lam. Automatic Extraction of Object-Oriented Component Interfaces. In *Proceedings of the International Symposium of Software Testing and Analysis*, pages 218–228, 2002.

[61] J. A. Whittaker and H. H. Thompson. *How to Break Software Security*. Addison Wesley, 2003.

[62] Y. Xie and A. Aiken. Scalable Error Detection Using Boolean Satisfiability. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, 2005.