Securing Web Applications with Static and Dynamic Information Flow Tracking

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

Benjamin Livshits
Microsoft Research
livshits@microsoft.com

John Whaley
Moka5, Inc.
jwhaley@moka5.com

Abstract

SQL injection and cross-site scripting are two of the most common security vulnerabilities that plague web applications today. These and many others result from having unchecked data input reach security-sensitive operations. This paper describes a language called PQL (Program Query Language) that allows users to declare to specify information flow patterns succinctly and declaratively. We have developed a static context-sensitive, but flow-insensitive information flow tracking analysis that can be used to find all the vulnerabilities in a program. In the event that the analysis generates too many warnings, the result can be used to drive a model-checking system to analyze more precisely. Model checking is also used to automatically generate the input vectors that expose the vulnerability. Any remaining behavior these static analyses have not isolated may be checked dynamically. The results of the static analyses may be used to optimize these dynamic checks.

Our experimental results indicate the language is expressive enough for describing a large number of vulnerabilities succinctly. We have analyzed over nine applications, detecting 30 serious security vulnerabilities. We were also able to automatically recover from attacks as they occurred using the dynamic checker.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification–Reliability

General Terms Security, Reliability

Keywords pattern matching, web applications, SQL injection, cross-site scripting, static analysis, dynamic analysis, model checking

1. Introduction

The security of Web applications has become increasingly important in the last decade. With more and more Web-based applications deal with sensitive financial and medical data, it is crucial to protect these applications from hacker attacks. A security assessment by the Application Defense Center, which included more than 250 Web applications from e-commerce, online banking, enterprise collaboration, and supply chain management sites concluded that at least 92% of Web applications are vulnerable to some form of attack [41]. Another survey found that about 75% of all attacks against Web servers target Web-based applications [18].

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

PEPM'08, January 7–8, 2008, San Francisco, California, USA.
ACM 978-1-59593-977-7/08/0001.

1.1 Information Tracking

Many vulnerabilities in web applications are caused by permitting unchecked input to take control of the application, which an attacker will turn to unexpected purposes. *SQL injection* is one of the top five external threats to corporate IT systems [37]. Via SQL injection, an attacker can introduce additional conditions or commands to a database query, thus allowing the attacker to bypass authentication or even alter or destroy data. In *cross-site scripting* (XSS), one of the top vulnerabilities in the past two years [6, 31], an attacker can trick a victim into clicking on a URL that takes over the browser. In the so-called "reflection attack" [14] XSS is used by a phisher to inject credential-stealing code into official sites without having to actually mimic the site he hopes to penetrate. SQL injection and XSS are but two *taint-based vulnerabilities* which can be detected by tracking the flow of untrusted data entered by the user and seeing if it flows unsafely into security-critical operations.

While this paper focuses on the use of information flow for securing web applications, the techniques described are useful for other topics such as debugging and avoiding leakage of confidential data. It is a general concept that has been used not only at the programming level [27], but also in operating systems [44] and hardware [11]. It is highly desirable that programmers be able to specify the information flow of interest simply in a high-level language and have tailored, sophisticated analyses automatically generated to detect such flow. In this way, programmers are able to leverage sophisticated analyses without being program analysis experts themselves.

Information tracking is a program property that requires new language support; using traditional techniques, such as program assertions or type declarations, to track information would require many lines of specifications sprinkled throughout the source code.

1.2 PQL

This paper describes a high-level declarative language called PQL (Program Query Language) [26]. PQL allows programmers to describe a class of information flow as a pattern that resembles an excerpt of Java code. For example, we can specify a simple SQL injection, which involves the flow of input data to database command routines, in just a few lines of PQL. Our system automatically detects the existence of information flow in a program matching a specified pattern both statically and dynamically, In addition, the programmer can specify the corrective actions to take if such a pattern is detected dynamically. In this way, the program heals automatically rather than simply reporting an error and terminating the program. This auto-healing property is important to prevent users from mounting a denial-of-service attack by crashing the program.

1.3 Integrating Static and Dynamic Analyses

From a PQL query, our system automatically generates a set of complementary static and dynamic analyses to detect matches to the information flow specification. Our system leverages three kinds of analysis techniques to help users track information flow, as shown in the overview in Figure 1.

- *Sound static information trackers using context-sensitive pointer alias analysis*. Sound information flow analysis is challenging because objects carrying the information may be passed around as heap references and method parameters throughout the program. We have developed an accurate information tracking analysis based on a context-sensitive, flow-insensitive pointer alias analysis [43]. PQL queries are systematically translated into Datalog queries, a logic programming language for deductive databases [36]. We use the bddbddb (Binary-Decision-Diagram Based Deductive DataBase) system to translate Datalog queries into BDD operations–the BDD representation makes it possible to encode the exponentially many calling contexts in large Java applications succinctly [42]. We also use the pointer alias analysis to help resolve reflection accurately. We have shown in some cases that the analysis is strong enough to identify all the possible vulnerabilities [25]. In cases where static analysis generates too many potential errors to analyze, further steps are taken as discussed below.
- *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. PQL transcends traditional syntax-based approaches by matching against the history of events witnessed by object instances. The higher-level semantics of PQL enables the use of static analysis to reduce the overhead of dynamic checking.
- *Model checking.* We next apply a more accurate static analysis, model checking, to the instrumented byte code to refine information tracking. By simulating the program execution against all inputs, model checking has the potential of producing a complete set of *attack vectors*—inputs that trigger a match to the vulnerability pattern. Unlike the context-sensitive flow-insensitive analysis, this checker produces no false positives. The programmer is presented with the exact input vector that causes the vulnerability and a succinct record of the events that lead to the attack. They give the programmer the incentive and the assistance to correct the program accordingly.

  Applying model checking to real-life applications is challenging because it is hard to analyze the large number of possible execution sequences for all possible inputs. As a result, many have resorted to model checking an abstract model, which unfortunately may not accurately reflect the behavior of the program. We show that we can apply model checking concretely, on a large number of web applications, abstracting only the user and the support libraries. The key is to analyze data dependencies in the application so that we can focus the model checkers on inputs that are likely to lead to a match.
- *Dynamic error recovery.* Finally, if static analysis cannot guarantee the code to be safe, we resort to monitoring the program's information flow dynamically. 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.

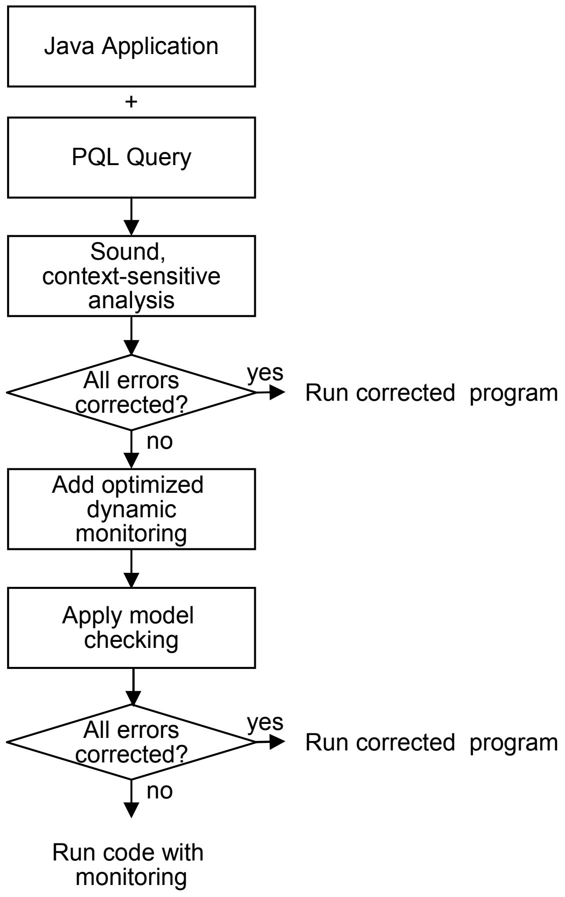

Figure 1. Overview of the PQL system

We have implemented all the algorithms presented in this paper, and have applied them to a large set of open-source web applications. We detected many vulnerabilities statically, and were able to effectively defend applications against attacks via dynamic detection and response [25, 26].

1.4 Organization

The rest of the paper is organized as follows. Section 2 discusses the nature of the security threats we counter. Section 3 discusses PQL, the specification language we use to drive our analyses. Sections 4, 5, and 6 discuss our techniqures. Section 7 provides a sample of our experimental results. Section 8 discusses related work, and Section 9 concludes.

2. Security Vulnerability Patterns

This section starts by presenting SQL injection, one of the most popular web vulnerabilities, then gives a high-level description of other vulnerabilities. It also introduces our Program Query Language by showing how a query for one of those vulnerabilities is expressed in PQL.

2.1 SQL Injection

SQL injection vulnerabilities are caused by unchecked user input propagating to a database for execution. A hacker may be able to embed SQL commands into an SQL query the application passes

```
    query simpleSQLInjection()
    var object String p;
    matches {
        p = HttpServletRequest.getParameter(_);
        Connection.execute(p);
    }
```

Figure 2. Simple SQL injection query.

to the database for execution. These unauthorized commands may view, update, or delete records. This type of vulnerability is especially critical in Web applications exposed to a large audience; any vulnerabilities at all mean that database information may be forged or stolen by anyone.

Let us look at a simple, concrete example. Here is a code fragment that may be found in a Java servlet hosting a Web service:

```
String p = request.getParameter(_);
con.execute(p);
```

This code reads a parameter from an HTTP request and passes it directly to a database back-end. By supplying a properly crafted query, a malicious user can gain unauthorized access to 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 if there exist some

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

in some possible run of the program 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. Note that these two events need not happen consecutively; the string p can be passed around as a parameter or stored on the heap before it is eventually used.

The PQL language allows us to describe this pattern simply, as shown in Figure 2. PQL queries are expressed as a pattern of dynamically executed statements. The statements listed in the query form a regular expression (in this case, a simple sequence of two method invocations) and the variables represent parameterized objects. Variables whose values are immaterial (such as r and c) can be specified purely by type name. Variables whose values and types are both immaterial are represented by the "don't care" symbol "_". Conceptually, these statements represent the smallest piece of code that could produce the behavior we are searching for. We do not care about any statements that may occur between them, nor do we care precisely how the objects are named in the code.

It is easy to see how we can translate this to a static analysis. We are looking for two static statements

```
p1 = r.getParameter("query");
```

and

```
c.execute(p2);
```

such that `p1` and `p2` may point to the same object.

In general, SQL injections are more subtle and require more sophisticated patterns to detect. In particular, the contents of the string are typically processed in some way, often inserted into a preformed query, before being passed to the database for execution.

Figure 3 gives a more complete PQL query for SQL injections. The main query binds the variable `source` to an initial input drawn from an HTTP request, then binds `tainted` to any value reachable with zero or more derivation steps. This is handled via the

```
query main(object Object source, object Object tainted)
matches {
    source = HttpServletRequest.getParameter();
    derivedString(source, tainted);
    java.sql.Statement.execute(tainted);
}

query derivedString(object Object x, object Object y)
var object Object temp;
matches
      y := x
    | { temp.append(x); derivedString(temp, y); }
    | { temp = x.toString(); derivedString(temp, y); }
```

Figure 3. A more complete SQL injection query in PQL.

`derivedString` subquery, a tail recursive loop that tracks derivation through the functions involved in string concatenation. Once a tainted object has been identified, it then searches for its use by the database. This query can similarly be translated into a static analysis.

2.2 Taint-Based Vulnerabilities

SQL injection is but one of many widespread vulnerabilities caused by unchecked input in today's Web-based systems. These problems can be generalized to *taint-based vulnerabilities*, which are specified by a set of *sources*, *sinks*, and *derivation methods*. Sources are methods that return data obtained from user input. Sinks are the sensitive methods that should not have tainted data passed in. Derivation methods specify how tainted data propagates from one object to another. By varying the composition of these sets, one can express all the vulnerabilities mentioned below.

Cross-site scripting is an attack on applications that fail to filter or quote HTML metacharacters in user input used in dynamically generated Web pages. Typically, the attacker tricks the victim into visiting a trusted URL containing a cross-site scripting vulnerability. This allows the attacker to embed malicious JavaScript code into the dynamically generated page and execute the script on the machine of any user that views the page [7]. When executed, malicious scripts may hijack the user's account, change the user's settings, steal the user's cookies, or insert unwanted content (such as ads) into the page.

HTTP response splitting is an attack on applications that fail to filter or quote newlines in header information. It enables various other attacks such as Web cache poisoning, cross user defacement, hijacking pages with sensitive user information, and cross-site scripting [21]. The crux of the HTTP response splitting technique is that the attacker may cause two HTTP responses to be generated in response to one maliciously constructed request. For HTTP splitting to be possible, the vulnerable application must include unchecked input as part of a response header sent back to the client.

Path traversal vulnerabilities allow a hacker to access or control files outside of the intended path [31, 40]. They occur when applications use unchecked user input in a path or file name; input normally arrives via URL input parameters, cookies, or HTTP request headers. Often the file in question is part of an ad-hoc database, for instance an image in a theme. In addition to reading or removing sensitive files, the attacker may attempt a denial-of-service attack by causing the application to access a file for which it does not have permissions.

All taint-based vulnerabilities, and many other error patterns, can be expressed easily in PQL. PQL allows programmers to use a familiar Java syntax to track operations applied to objects simply, regardless of how these objects are referred to in the program text.

3. 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.

3.1 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 occurs. 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 4 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 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, and `con` is bound to object o_5.

This execution demonstrates two SQL injection vulnerabilities. The first match is demonstrated with unchecked user data flowing from o_3 to o_5, and again from o_7. □

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

Figure 4. Abstract execution trace for Example 1.

3.2 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 5. 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.

3.2.1 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.

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).

3.2.2 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

```
queries       ⟶  query*

query         ⟶  query qid ( [decl [, decl]*] )
                 [var 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
                 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 5. BNF grammar specification for PQL.

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.

3.2.3 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`.

3.2.4 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.

4. Context-Sensitive Static Information Tracking

We have developed an algorithm to automatically translate PQL queries into queries utilizing the results of a context-sensitive pointer analysis. This shields 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.

Our checkers use pointer information from a sound cloning-based context-sensitive inclusion-based pointer alias analysis due to Whaley and Lam [43]. 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. We can then use bddbddb to resolve the queries.

Datalog is highly expressive and includes the ability to recursively specify properties, meaning that PQL queries may be translated to Datalog approximation using a simple syntax-directed approach.

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.

After running bddbddb, we will have as our result a set of program objects that could participate in the match of each subquery.

This information may be presented to the application programmer for inspection. The tool will report *all* the potential errors, some of which may not be errors in the program. The application programmer can determine if the error is a true vulnerability, which would require a fix, or it is a result of the analysis's imprecision. If the programmer can fix all the bugs, there is no need to perform any more analysis.

In the case further analysis is necessary, the results are used to reduce the amount of code that needs to be monitored. Because our analysis is sound, any program point that our analysis does *not* flag as suspicious is guaranteed to never participate in a match. Thus, when modifying the binary to check dynamic results, we need not instrument any point not present in the reported program points.

5. Dynamic Monitoring Code

PQL automatically generates a specialized matcher for a query and weaves instrumentation into the target application to perform the match at runtime.

A direct, naïve approach to finding matches to PQL queries dynamically would consist 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.
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. We use two main strategies to lower the overhead. First, we modify the program to only track objects at 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 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.

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.

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.

6. Model Checking

Model checking is attractive for web security because not only can it find errors, it can be used to generate the attack vectors to prove the existence of a real vulnerability. This information is very helpful for the programmers to fix the bugs. However, model checking is challenging for real programs. A web application consists of tens or hundreds of thousands of lines of code. It continuously accepts inputs, so it is impossible to exhaust all possible paths.

Web developers have greatly leveraged common frameworks to reduce the development time for creating web applications. We show that we can exploit the use of common frameworks to produce powerful programming tools. We have developed a model checker, called QED (Query-based Event Director). This model checker is designed for web applications built on top of servlets, JSPs, and Apache Struts:

- Java servlets [34], which is a standard extension to the Java platform for writing web applications.
- JSPs (Java Server Pages) [35], which allow page design to be commingled with database accesses.
- Apache Struts [2], which is a web application framework that uses the model-view-controller paradigm. In this paradigm, a controller decouples the data model from the user view so they can easily be changed independently.

An attack vector is a sequence of URLs, each of which consists of a page request and a set of input parameters. One of the strengths of QED is to be able to find attack vectors that consist of more than one URL request quickly. Instead of generating large number of random input vectors to exercise the program, QED uses a *goal-directed* approach to generate those essential input vectors that exercise those portions of code that may harbor vulnerabilities.

The input to QED is the Java bytecode instrumented to detect the vulnerability patterns of interest. The monitoring code has been optimized so only those sections that can participate in an attack, according to the context-sensitive information tracking analysis, are instrumented.

Each URL request generated leads to the invocation of an event. By understanding how event handling code is dispatched in the Java servlet framework, it can deduce the URLs that need to be supplied to exercise instrumented code. A URL request is redundant unless it can generate or lead to an event that contains instrumented code. This dependence analysis allows QED to prune off unnecessary input sequences, focusing on short, highly productive input vectors.

QED uses Java Pathfinder [38] to perform model checking on instrumented code. QED stubs out various components in the application. For example, for the sake of efficiency, we use non-deterministic choices reflecting the different paths we wish to explore instead of executing a full object persistence layer. These preparations can be re-used by appplications using the same modules.

In summary, QED takes advantage of the high-level semantics of the Java servlet framework, the high-level PQL query, and the bddbddb static analysis system to provide an effective model checker for finding taint-based vulnerabilities in a large number of web-based applications.

7. Experimental Results

We have applied PQL to describe a large of security flaws, including SQL injection and cross-site scripting. Besides security vulnerabilities we have also used PQL to find other kinds of program errors such as violations of consistency invariants and resource leaks. We have applied PQL to over 60,000 Java classes and found over 200 errors.

7.1 SQL Injection

We first show an example of a real-life vulnerability query written in PQL. Figure 6 contains an excerpt of the SQL injection query for web applications written in the J2EE framework. Sources, listed in query `UserSource` 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 `String`s. It invokes the `StringProp` query, which handles all the ways in which one string can be derived from another. By modifying the start and end points of the information flow, we produced PQL queries for each of the vulnerabilities discussed in Section 2.

7.2 Context-Sensitive Information Tracking

We applied each of our queries to a set of representative open-source applications: `jboard`, `blojsom`, `snipsnap`,

```
query main(object Object source, object Object sink)
var
    object java.sql.Connection con;
    object java.sql.Statement stmt;
matches {
    UserSource(source);
    StringPropStar(source, sink);
} 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, object Object y)
matches
      y.append(x)
    | y = new String(x)
    | y = new StringBuffer(x)
    | y = x.toString()
    | ...

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


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

Figure 6. Full SQL injection query.

`personalblog`, `pebble` and `blueblog` are Web-based bulletin board and blogging applications; `webgoat` is a J2EE application designed as a test case and teaching tool; and `road2hibernate` is a test program for the popular object persistence library `hibernate`.

The results of the experiment are shown in Figure 7. We found that every application suffers from one or more vulnerabilities we tested, except for the smallest application, `jboard`. `snipsnap` is the only application that suffers from the HTTP splitting vulnerability; it has eleven such errors. Path traversal vulnerabilities are found in two applications, whereas potential SQL injection and cross-site scripting errors are located in four applications. In total, our experiment turned up 30 errors for further investigation.

7.3 Model Checking

The static analysis results are given in terms of objects or program points. In order to automatically derive interactions with a web application that produce the attack, more work is needed. The model-checking system can fabricate attacks based on the program points in the result. This process produced concrete attack vectors for the Cross-Site scripting vulnerabilities in `personalblog`.

7.4 Dynamic Information Tracking

It is important to keep the runtime overhead low as we track the information and catch security vulnerabilities dynamically. Without static optimization, many program locations need to be instrumented. For example, 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. The model checker relies on instrumented code to develop its attacks, so excessive instrumentation is also a problem for it.

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

Figure 7. Vulnerabilities found in 9 Web applications.

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. The reduction in the number of instrumentation points due to static optimization can be as high as 97% in `roller` and 99% in `personalblog`. This 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.

7.5 Auto-healing

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 unsuitable.

We can use PQL to check for potential errors and recover gracefully. The query shown in Figure 6 uses a `replaces` clause to check if the inputs are clean, and if not, replace it with a safe version.

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`.

8. Related Work

Web applications carry a unique set of security risks [31]. Various systems have been developed to help secure web applications. SABER [32] is a static tool that detects a large number of common design errors based on instantiations of a number of error pattern templates. WebSSARI [17] and Nguyen-Tuong et al. [29] 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 much finer granularity. In a more general context FindBugs [16] attempts to locate a broad class of bugs in Java applications of all kinds.

The SQLCHECK system [33] uses a much more precise technique to detect grammatical changes in commands as a result of user input. while SQLCHECK is SQL-injection specific and FindBugs is a battery of unrelated analyses. Taint flow within an application is tracked incidentally, and only if the PQL specification demands it.

To combat cross-site scripting, recent work has focused on extending the DOM to permit browser extensions to block out any unauthorized scripts [20]. While, if fully implemented, this system will block out any possible attacks, it requires cooperation between both site authors and clients.

8.1 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 [5]; later tools have expanded on the basic concept to provide additional power. Dalek [30] 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.

PQL's dynamic monitoring system also bears a certain resemblance to *aspect-oriented programming*, in which the programmer specifies locations and conditions under which extra actions must be taken. Walker and Veggers [39] introduce the concept of *declarative event patterns*, in which regular expressions of traditional syntactic pointcuts are used to specify when advice should run. Allan et al. [1] extend this further by permitting PQL-like free variables in the patterns.

In an alternative approach, the Partiqle system [12] 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.

8.2 Other Program Query Languages

Systems like ASTLOG [10] and JQuery [19] permit patterns to be matched against source code; Liu et al. [24] extend this concept to include parametric pattern matching [3]. 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 [22]. 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.

8.3 Analysis Generators

PQL follows in a tradition of powerful tools that take small specifications and use them to automatically generate analyses. Metal [13] and SLIC [4] 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 [23] 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 [28] also presents a general technique for dynamic analyses on binaries.

8.4 Model Checkers

Model checking systems such as SPIN [15] are powerful and widespread tools for capturing complicated program properties. Model checkers generally operate upon abstract languages such as Promela; the Bandera project [8] 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 [9].

Our system uses the Java PathFinder system [38]. JPF was suitable for our system primarily due to its ability to directly run sizable Java applications as bytecode; this permitted us to treat our dynamic analysis as just another part of the application being checked.

9. Conclusion

Information flow is one of the basic analyses used in compiler optimizations; as such it is usually applied only to local variables within a procedure. Whole-program information flow of dynamically allocated objects is necessary for higher level software engineering tools. This paper describes one particularly compelling use, web application security, where information flow information is directly and immediately applicable. We show that we can automatically find a large number of security vulnerabilities in real-life web applications through the synergism of a new language for describing information flow, context-sensitive pointer alias analysis, dynamic monitoring, and model checking.

The results presented in this paper are likely to be just the beginning of several important trends.

1. Information flow tracking is likely to be used in many more applications. Security vulnerabilities in the form of untrusted user input flow to security critical operations is just one example. Another important use is the prevention of leakage of confidential information. As information flow is essential to general understanding of a program, it will make possible more sophisticated software engineering tools.
2. Static information flow tracking at the object level is now possible for real-life applications thanks to a powerful context-sensitive pointer alias analysis. We showed that it can be used directly to prove certain facts like the absence of security vulnerabilities in a program, or to reduce the overhead of model checking and dynamic analysis. This is just the beginning, as improvements are still needed to improve the accuracy and scalability of static context-sensitive analysis.
3. The use of components, libraries, frameworks has greatly accelerated the development of software. We expect that more framework-specialized tools, such as the QED model checker, will be developed to help programmers with not just the development, but the debugging and evolution of the software.
4. Just as high-level programming languages have greatly improved software productivity, high-level languages like PQL that help programmers with the complete software life cycle will have an important impact. PQL, Datalog, BDD libraries, and Java Pathfinder all provide powerful abstractions. The translation between the layers is straightforward thus rendering the system robust. At the same time, the implementation of each layer shields a significant level of details from the users. Our experimental results demonstrate that it is possible now for relatively naïve users to create custom program analyses involving sophisticated components.

References

[1] 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*, pages 345–364, 2005.

[2] Apache Software Foundation. Apache Struts. `http://struts.apache.org`, 2002.

[3] 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.

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

[5] 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.

[6] S. M. Christey. Vulnerability type distribution in CVE. `http://www.attrition.org/pipermail/vim/2006-September/001032.html`.

[7] S. Cook. A Web developers guide to cross-site scripting. `http://www.giac.org/practical/GSEC/Steve_Cook_GSEC.pdf`, 2003.

[8] 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 *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, 2000.

[9] 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.

[10] 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.

[11] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*, pages 482–493, 2007.

[12] 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.

[13] 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.

[14] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley Publishing, 2004.

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

[16] 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)*, pages 132–136, 2004.

[17] 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.

[18] G. Hulme. New software may improve application security. `http://www.informationweek.com/story/IWK20010209S0003`, 2001.

[19] 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.

[20] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proceedings of the 16th International World Wide Web Conference (WWW'07)*, pages 601–610, 2007.

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

[22] 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.

[23] 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.

[24] 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.

[25] V. B. Livshits and M. S. Lam. Finding Security Errors in Java Programs with Static Analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271–286, Aug. 2005.

[26] M. C. Martin, B. Livshits, and M. S. Lam. Finding Application Errors and Security Flaws using PQL: a Program Query Language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 365–383, 2005.

[27] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[28] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100, 2007.

[29] 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 (SEC)*, pages 295–308, 2005.

[30] 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.

[31] OWASP. The ten most critical web application security vulnerabilities. `http://www.owasp.org/images/e/e8/OWASP_Top_10_2007.pdf`, 2007.

[32] 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*, pages 243–251, 2004.

[33] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *POPL '06: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–382, 2006.

[34] Sun Microsystems. JSR-000154 Java Servlet 2.5 Specification. `http://jcp.org/aboutJava/communityprocess/mrel/jsr154/index.html`, 2004.

[35] Sun Microsystems. JSR-000245 JavaServer Pages 2.1. `http://jcp.org/aboutJava/communityprocess/final/jsr245/index.html`, 2006.

[36] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, Md., volume II edition, 1989.

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

[38] W. Visser, K. Havelund, G. Brat, S.-J. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[39] 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.

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

[41] WebCohort, Inc. Only 10% of Web applications are secured against common hacking techniques. `http://www.imperva.com/company/news/2004-feb-02.html`, 2004.

[42] J. Whaley. bddbddb: BDD-Based Deductive DataBase. `http://bddbddb.sourceforge.net`, 2004.

[43] 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.

[44] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieères. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 263–278, 2006.