SecuriFly:
Runtime Protection and Recovery from Web Application Vulnerabilities

Benjamin Livshits, Michael Martin, and Monica S. Lam

Technical Report
Stanford University

September 22, 2006

Abstract

This reports presents a runtime solution to a range of Web application security vulnerabilities. The solution we proposes called SecuriFly consists of instrumenting the application to precisely track the flow of data. When a potential vulnerability is observed, the application is either terminated to prevent the vulnerability from being exploited or special recovery code is executed and the application is allowed to continue on running. We have used SecuriFly to harden and experiment with a range of large open-source benchmarks written in Java. Protection provided by SecuriFly was sufficient to protect against all exploits we were able to generate.

Chapter 1

Introduction

The landscape of security vulnerabilities has changed dramatically in the last several years. While buffer overruns and format string violations accounted for a large fraction of all exploited vulnerabilities in the 1990s, the picture started to change in the first decade of the new millennium. As Web-based applications became more prominent, familiar buffer overruns are now far outnumbered by *Web application vulnerabilities* such as SQL injections and cross-site scripting attacks.

In this report, we introduce SecuriFly, which provides a comprehensive runtime compiler-based solution to a wide range of Web application vulnerabilities. Our approach targets large real-life Web-based Java applications.

Given a vulnerability description, specially instrumented, secured application bytecode is produced. To make our approach both extensible and user-friendly, vulnerability specifications are expressed in PQL, a Program Query Language. The initial PQL vulnerability specification is provided by the user, but most of the specification can be shared among multiple applications being analyzed. Secured executables may be deployed on a standard application server. Furthermore, to improve application uptime, vulnerability recovery rules may be specified. Finally, we show how static analysis can be used to significantly reduce the instrumentation overhead.

1.1 Overview of Web Application Vulnerabilities

Of all vulnerabilities identified in Web applications, problems caused by *unchecked input* are recognized as being the most common [Ope04]. To exploit unchecked input, an attacker needs to achieve two goals:

Inject malicious data into Web applications. Common methods used include:

- **Parameter tampering**: pass specially crafted malicious values in fields of HTML forms.
- **URL manipulation**: use specially crafted parameters to be submitted to the Web application as part of the URL.
- **Hidden field manipulation**: set hidden fields of HTML forms in Web pages to malicious values.
- **HTTP header tampering**: manipulate parts of HTTP requests sent to the application.
- **Cookie poisoning**: place malicious data in cookies, small files sent to Web-based applications.

Manipulate applications using malicious data. Common methods used include:

- **SQL injection**: pass input containing SQL commands to a database server for execution.
- **Cross-site scripting**: exploit applications that output unchecked input verbatim to trick the user into executing malicious scripts.
- **HTTP response splitting**: exploit applications that output input verbatim to perform Web page defacements or Web cache poisoning attacks.
- **Path traversal**: exploit unchecked user input to control which files are accessed on the server.

- **Command injection**: exploit user input to execute shell commands.

These kinds of vulnerabilities are widespread in today's Web applications. A recent empirical study of vulnerabilities found that parameter tampering, SQL injection, and cross-site scripting attacks account for more than a third of all reported Web application vulnerabilities [SS04]. While different on the surface, all types of attacks listed above are made possible by user input that has not been (properly) validated. This set of problems is similar to those handled dynamically by the *taint mode* in Perl [WCS96], even though our approach is considerably more extensible. We refer to this class of vulnerabilities as the *tainted object propagation* problem. Detailed information about these classes of vulnerabilities can be found in "The 21 Primary Classes of Web Application Threats" [Net04a] and the "OWASP Secure Development Guide [Ope05]".

In this section we focus on a variety of security vulnerabilities in Web applications that are caused by unchecked input. According to an influential survey performed by the Open Web Application Security Project [Ope04], unvalidated input is the number one security problem in Web applications. Many such security vulnerabilities have recently been appearing on specialized vulnerability tracking sites such as `SecurityFocus` and were widely publicized in the technical press [Net04a, Ope04]. Recent reports include SQL injections in Oracle products [Lit03a] and cross-site scripting vulnerabilities in Mozilla Firefox [Kra05].

1.1.1 SQL Injection Example

Let us start with a discussion of SQL injections, one of the most well-known kinds of security vulnerabilities found in Web applications. SQL injections are caused by unchecked user input being passed to a back-end database for execution [Anl02a, Anl02b, Fri04, Kos04, Lit03b, Spe02b]. The hacker may embed SQL commands into the data he sends to the application, leading to unintended actions performed on the back-end database. When exploited, a SQL injection may cause unauthorized access to sensitive data, updates or deletions from the database, and even shell command execution.

Example 1.1. A simple example of a SQL injection is shown below:

```
HttpServletRequest request = ...;
String userName = request.getParameter("name");
```

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

This code snippet obtains a user name (`userName`) by invoking method `request.getParameter("name")` and uses it to construct a query to be passed to a database for execution (via `con.execute(query)`). This seemingly innocent piece of code may allow an attacker to gain access to unauthorized information: if an attacker has full control of string `userName` obtained from an HTTP request, he can for example set it to `'OR 1 = 1;−−`. Two dashes are used to indicate comments in the Oracle dialect of SQL, so the `WHERE` clause of the query effectively becomes the tautology `name = '' OR 1 = 1`. This allows the attacker to circumvent the name check and get access to all user records in the database. □

SQL injection is but one of the vulnerabilities that can be formulated as *tainted object propagation* problems. In this case, the input variable `userName` is considered *tainted*. If a tainted object (the *source* or any other object derived from it) is passed as a parameter to `con.execute` (the *sink*), then there is a vulnerability. As discussed above, such an attack typically consists of two parts: (1) injecting malicious data into the application and (2) using the data to manipulating the application. The former corresponds to the *sources* of a tainted object propagation problem and the latter to the *sinks*. The rest of this section presents attack techniques and examples of how exploits may be created in practice.

1.1.2 Injecting Malicious Data

Protecting Web applications against unchecked input vulnerabilities is difficult because applications can obtain information from the user in a variety of different ways. One must check all sources of user-controlled data such as form parameters, HTTP headers, and cookie values systematically. While commonly used, client-side filtering of malicious values is not an effective defense strategy. For example, a banking application may present the user with a form containing a choice of only two account numbers; however, this restriction can be easily circumvented by saving the HTML page, editing the values in the list, and resubmitting the form. Therefore, inputs must be filtered by the Web application on the server. Note that many attacks are relatively easy to mount: an attacker needs little more than a standard Web browser to attack Web applications in most cases.

Parameter Tampering

The most common way for a Web application to accept parameters is through HTML forms. When a form is submitted, parameters are sent as part of an HTTP request. An attacker can easily tamper with parameters passed to a Web application by entering maliciously crafted values into text fields of HTML forms.

URL Tampering

For HTML forms that are submitted using the HTTP `GET` method, form parameters as well as their values appear as part of the URL that is accessed after the form is submitted. An attacker may directly edit the URL string, embed malicious data in it, and then access this new URL to submit malicious data to the application.

Example 1.2. Consider a Web page at a bank site that allows an authenticated user to select one of her accounts from a list and debit $100 from the account. When the submit button is pressed in the Web browser, the following URL is requested:

```
http://www.mybank.com/myaccount?accountnumber=341948&debit_amount=100
```

However, if no additional precautions are taken by the Web application receiving this request, accessing

```
http://www.mybank.com/myaccount?accountnumber=341948&debit_amount=-5000
```

may in fact increase the account balance. □

There are other URL parameters that an attacker can modify, including attribute parameters and internal modules. Attribute parameters are unique parameters that characterize the behavior of the uploading page. For example, consider a content-sharing Web application that enables the content creator to modify content, while other users can only view content. The Web server checks whether the user that is accessing an entry is the author or not (usually by cookie). An ordinary user will request the following link:

```
http://www.mydomain.com/myaccount?id=77492&mode=readonly
```

An attacker can modify the mode parameter to `readwrite` in order to gain authoring permissions for the content.

Hidden Field Manipulation

Because HTTP is stateless, many Web applications use hidden fields to emulate persistence. Hidden fields are just form fields made invisible to the end-user. For example, consider an order form that includes a hidden field to store the price of items in the shopping cart:

```
<input type="hidden" name="total_price" value="25.00">
```

A typical Web site using multiple forms, such as an online store will likely rely on hidden fields to transfer state information between pages. For instance, a single page we sampled on `Amazon.com` contains a total of 25 built-in hidden fields. Unlike regular fields, hidden fields cannot be modified directly by typing values into an HTML form. However, since the hidden field is part of the page source, saving the HTML page, editing the hidden field value, and reloading the page will cause the Web application to receive the newly updated value of the hidden field. This attack technique is commonly used to forge information being sent to the Web application and to mount SQL injection or cross-site scripting attacks.

HTTP Header Manipulation

HTTP headers typically remain invisible to the user and are used only by the browser and the Web server. However, some Web applications do process these headers, and attackers can inject malicious data into applications through them. While a normal Web browser will not allow forging the outgoing headers, multiple freely available tools allow a hacker to craft an HTTP request leading to an exploit [Chi04].

Example 1.3. An HTTP request fragment is shown below:

```
Host: www.mybank.com
Accept-Language: en-us, en;q=0.50
User-Agent: Lynx/2.8.4dev.9 libwww-FM/2.14
Referer: http://www.mybank.com/login
Content-type: application/
              x-www-form-urlencoded
Content-length: 100
```

```
con.executeUpdate("UPDATE EMPLOYEES "
    + " SET SALARY = " + salary
    + " WHERE ID = " + id);
```

(a)

```
PreparedStatement pstmt =
     con.prepareStatement(
        "UPDATE EMPLOYEES " +
        " SET SALARY = ? " +
        " WHERE ID = ?");

pstmt.setBigDecimal(1, salary);
pstmt.setInt(2, id);
```

(b)

Figure 1.1: Two different ways to update an employee's salary: (a) may lead to a SQL injection and (b) safely updates the salary using a `PreparedStatement`.

The `Accept-Language` header indicates the preferred language of the user. An internationalized Web application may take the language label from the HTTP request and pass it to a database to look up a language-specific text message. If the this header is sent *verbatim* to the database, an attacker may inject SQL commands by modifying the header value. Likewise, if the header value is used to build a file name with messages for the correct language, an attacker may be able to launch a path-traversal attack [Ope05]. □

Consider, for example, the `Referer` field, which contains the URL indicating where the request comes from. This field is commonly trusted by the Web application, but can be easily forged by an attacker. It is possible to manipulate the `Referer` field's value used in an error page or for redirection to mount cross-site scripting or HTTP response splitting attacks. Similarly, the `Referer` field should never be used to authenticate valid clients, as this authentication scheme may be easily circumvented [Ope05].

Cookie Poisoning

Cookie poisoning attacks consist of modifying a cookie, which is a small file accessible to Web applications stored on the user's computer [Kle02b]. Many Web applications use cookies to store information such as user login/password pairs and user identifiers. This information is often created and stored on the user's computer after the initial interaction with the Web application, such as visiting the application login page. Cookie poisoning is a variation of header manipulation: malicious input can be passed into applications through values stored within cookies. Because cookies are supposedly invisible to the user, cookie poisoning is often more dangerous in practice than other forms of parameter or header manipulation attacks.

Example 1.4. Consider the HTTP `GET` request in Figure 1.2. The URL on host `http://www.mybank.com` requested by the browser transfer and the parameter string `transfer` = `yes` indicates that the user wants to perform a funds transfer.

The request includes a cookie that contains the following parameters: `SESSION`, which is a unique identification string that associates the user with the site and `Amount`, which is the transfer amount for this transaction. `Amount` is validated by the Web application before being stored in a cookie. However, an attacker can easily edit the cookie and change the `Amount` value in order to circumvent account overdraw checks that are performed before the cookie is created to transfer more money that is contained in an account. □

As this example illustrates, cookie poisoning is typically used in a manner similar to hidden field manipulation, i.e. to change the outcome the attacker's advantage. However, since programmers rely on cookies as a location for storing parameters, all parameter attacks including SQL injection, cross-site scripting, etc. can be performed with the help of cookie poisoning [Bar03].

Non-Web Input Sources

Malicious data can also be passed in as command-line parameters. This problem is not as important because typically only administrators are allowed to execute components of Web-based applications directly from the command line. However, by examining our benchmarks, we discovered that command-line utilities are often used to perform critical tasks such as initializing, cleaning, or validating a back-end database or migrating the data. Therefore, attacks against these important utilities can still be dangerous.

```
GET transfer?complete=yes
HTTP/1.0 Host: www.mybank.com Accept: */*
Referrer: http://www.mybank.com/login
Cookie: SESSION=89DSSSXX89JJSYUJG; Amount=5000
```

Figure 1.2: An HTTP `GET` request containing a cookie.

1.1.3 Exploiting Unchecked Input

Once malicious data is injected into an application, an attacker may use one of many techniques to take advantage of this data, as described below.

SQL Injections

SQL injections first described in Section 1.1.1 are caused by unchecked user input being passed to a back-end database for execution. When exploited, a SQL injection may cause a variety of consequences from leaking the structure of the back-end database to adding new users, mailing passwords to the hacker, or even executing arbitrary shell commands.

Many SQL injections can be avoided relatively easily with the use of better APIs. J2EE provides the `PreparedStatement` class, that allows specifying a SQL statement template with ?'s indicating statement parameters. Prepared SQL statements are precompiled, and expanded parameters never become part of executable SQL. However, not using or improperly using prepared statements still leaves plenty of room for errors.

Example 1.5. Figure 1.1 shows two ways to update the salary of an employee, whose id is provided. The first method in Figure 1.1 (a) uses string concatenation to construct the query and leading to potential SQL injection attacks; the second in Figure 1.1 (b) uses `PreparedStatements` and is safe from SQL injection attacks. □

Most SQL injections we have encountered can be categorized as the result of not using `PreparedStatements` and constructing SQL statements directly. However, while a good practical strategy for most purposes when programming using J2EE, `PreparedStamtents` are not a panacea. As our practical experience with auditing for SQL injections shows, there are some legitimate reasons for using dynamically constructed SQL statements:

- SQL statements depend on the way the application is configured. For instance, SQL statements are often read from configuration files that are different depending on the back-end database being used.
- Only certain parts of SQL statements may be parameterized, for instance, an online store that performs a search depending on both the search criterion that corresponds to a database column, such as the name or the address will likely construct the SQL query using string concatenation.

- Improper use of `PreparedStatements`, i.e. using non-constant template strings for constructing prepared statements defeats the purpose of using them in the first place.

Cross-site Scripting Vulnerabilities

Cross-site scripting occurs when dynamically generated Web pages display input that has not been properly validated [CGI, Coo03, Hu04, Kle02a, Spe02a]. An attacker may embed malicious JavaScript code into dynamically generated pages of trusted sites. When executed on the machine of a user who views the page, these scripts may hijack the user account credentials, change user settings, steal cookies, or insert unwanted content (such as ads) into the page. At the application level, echoing the application input back to the browser verbatim enables cross-site scripting.

Example 1.6. A cross-site scripting attack leverages the trust the user has for a particular Web site, such as that of a financial institution, to perform malicious activities. Suppose a bank's online accounting system has an error page that displays input verbatim. An attacker may trick the legitimate user into following a benign-looking URL, which results in displaying an error page containing a malicious script. Suppose the script looks like the following:

```
<script>
    document.location =
        'http://www.attack.org/?cookies=' +
            document.cookie
</script>
```

When the error page is opened, the script will redirect the user's browser, while submitting the user's cookie to a malicious site in the meantime. □

HTTP Response Splitting

HTTP response splitting is a general technique that enables various new attacks including Web cache poisoning, cross-user defacement, sensitive page hijacking, as well as cross-site scripting [Kle04]. By supplying unexpected line break CR and LF characters, an attacker can cause *two* HTTP responses to be generated for *one* maliciously constructed HTTP request. The second HTTP response may be erroneously matched with the next HTTP request. By controlling the second response, an attacker can generate a variety of issues, such as forging or *poisoning* Web pages on a caching proxy server. Because the proxy cache is typically shared by many users, this makes the effects of defacing a page or constructing a spoofed page to collect user data even more devastating. For HTTP splitting to be possible, the application must include unchecked input as part of the response headers sent back to the client. For example, applications that embed unchecked data in HTTP `Location` headers returned back to users are often vulnerable.

Several HTTP splitting vulnerabilities in deployed software have been announced in recently, including two in Java applications. `SecurityFocus.com` bid ids 11413 and 11180. The latter one is in `snipsnap`, which is one of the benchmarks in our suite. A common coding pattern that makes Java applications vulnerable to HTTP response splitting is redirecting to user-defined URLs, as illustrated by this code snipped from one of our benchmark applications, `personalblog`:

```
request.sendRedirect(request.getParameter("referer"));
```

Path Traversal

Path-traversal vulnerabilities allow a hacker to access or control files outside of the intended file access path. Path-traversal attacks are normally carried out via unchecked URL input parameters, cookies, and HTTP request headers. Many Java Web applications use files to maintain an ad-hoc database and store application resources such as visual themes, images, and so on.

If an attacker has control over the specification of these file locations, then he may be able to read or remove files with sensitive data or mount a denial-of-service attack by trying to write to read-only files. Using Java security policies allows the developer to restrict access to the file system (similar to using `chroot` jail in Unix). However, missing or incorrect policy configuration still leaves room for errors. When used carelessly, IO operations in Java may lead to path-traversal attacks.

Example 1.7. The following code snippet we found in `blojsom` turns out to be not secure because `permlink` is under user control:

```
String permalinkEntry =
                    _blog.getBlogHome() +
                    category + permalink;
File blogFile = new File(permalinkEntry);
```

Changing `permlink` on the part of the attacker can be used to mount denial of service attacks when accessing non-existent files. □

Command Injection

Command injection (also sometimes referred to as "Stealth Commanding") involves passing shell commands into the application for execution. This attack technique enables a hacker to attack the server using access rights of the application. While relatively uncommon in Web applications, especially those written in Java, this attack technique is still possible when applications carelessly use functions that execute shell commands or load dynamic libraries.

1.2 Advantages of the Runtime Approach

Commonly used dynamic techniques such as application firewalls [Net04b] that rely on pattern-matching and monitor traffic flowing in and out of the application are often a poor solution for SQL injection or cross-site scripting attacks. Such techniques suffer from both false positives and false negatives.

In contrast, our runtime technique can detect all attacks of a particular kind because it precisely tracks how the data flows through the application. No false alarms are introduced because runtime instrumentation has perfect historical information about any piece of data. Moreover, our approach can gracefully recover from vulnerabilities before they can do any harm by sanitizing tainted input whenever necessary. There are some inherent advantages summarized below that the runtime analysis approach has over the static one:

Deployment-time security. Runtime analysis can be integrated with the server so that whenever a new Web application is added, it is instrumented automatically. This removes the risk associated with deploying "unfamiliar", potentially unsafe Web applications. This approach eliminates the "vulnerability window" that stems from the code changing without the static analysis tool being immediately rerun. Moreover, recovery from vulnerabilities can be provided by applying user-provided sanitization.

No need to change the development lifecycle. Unlike static tools, runtime technology can be used at organizations that lack a wellestablished static analysis or testing infrastructure as part of their development process. Trying to introduce a static analysis tool into such an organization is a difficult task, one that is likely to be met with reluctance from the developers.

No need for the source code. Unlike a static approach, runtime analysis does not require changes to the original program and does not need access to the source code. While static analysis is done at the bytecode level, *reporting* analysis results back to the user requires access to the source code. Runtime analysis can be especially advantageous when dealing with applications that rely heavily on libraries, whose source is unavailable. In those cases, the vulnerabilities that span library code cannot be easily reported. It can also be beneficial in an environment where the source code is unavailable for security or intellectual property reasons.

Avoids static analysis challenges. Finally, analyzing Web applications statically can be challenging because of the difficulty of call graph construction and reflection. Runtime analysis avoids these challenges altogether.

1.3 Report Organization

The rest of this report is organized as follows. Chapter 2 provides an overview of SecuriFly. Chapter 3 describes the runtime system. Chapter 4 summarizes the experimental results. Chapter 5 talks about related work.

Chapter 2

Overview

The user of SecuriFly specifies what constitutes a vulnerability. Specifications are expressed in PQL, a Program Query Language [MLL05]. PQL is a generic language that can be used to capture events that happen to objects, such as specific method calls being invoked with an object passed as a parameter or returned from a method. While PQL has been used to express a variety of queries for purposes ranging from debugging to finding optimization opportunities, in this report it is used to capture vulnerability queries.

Since most portions of vulnerability specification consist of J2EE library methods, and since the J2EE library is shared among most Java Web applications, the per-application specification effort in usually minor. Moreover, most vulnerabilities can be found with a "generic" specification that is specific to the Web application development framework such as J2EE or Apache Struts, which completely removes the need for user involvement. A very simple PQL query that captures only *some* SQL injection vulnerabilities is shown in Figure 2.1; more complete vulnerability queries are described below. This PQL query will locate all objects `param` which are returned from a call to `getParameter` and are passed into method `executeQuery`.

Our runtime technique works by instrumenting the existing application based on the PQL specification provided by the user to prevent vulnerabilities at runtime. In addition to *not* suffering from false positives, the runtime approach offers the following important benefits:

- **Keeps vulnerabilities from doing harm.** As discussed earlier, runtime analysis may be used in situations where the user is unwilling to

```
query verySimpleSQLInjection()
returns
    object String param;
uses
    object HttpServletRequest  req;
    object Connection          con;
matches {
    param   = req.getParameter(_);

    con.execute(param);
}
```

Figure 2.1: A very simple PQL query for finding SQL injections.

consider the false positives. It also applies when the source code is unavailable or cannot be changed. The runtime technique is of great practical value in stopping existing vulnerabilities from being exploited. For example, an application that has an output validation vulnerability that may lead to an information leak can be terminated before the leak actually occurs.

- **Can recover from exploits.** Since the right approach to fixing taint-style vulnerabilities in Web applications involves applying a data sanitizer, our dynamic technique automatically applies the appropriate sanitizer on the code execution paths that lack it. The runtime approach we describe can be used in the creation of a safe application server, which automatically secures the applications that are deployed on it. This gives the user a notion of continuous security.

- **No false positives and no false negatives.** Finally, the dynamic technique has full visibility into the runtime program behavior and therefore does not suffer from false alarms. The runtime protection is designed to detect and prevent any vulnerabilities matching the user-provided specification.

As with any runtime technique, an important consideration is the runtime overhead. Naïve instumentation generated based on the PQL specification incurs an overhead ranging from 40% to 120%. While Web-based applications are largely interactive in nature, the overhead is still undesirable. In

SECURIFLY, additional static information is computed to reduce the amount of runtime instrumentation that needs to be inserted.

This approach is very effective, as it reduces the number of instrumentation points by about 85%-99%. This reduces the overhead to less than 37%. For most benchmarks, the overhead is under 20%. The soundness of the static technique allows us to remove instrumentation points deemed unnecessary statically without jeopardizing the quality of runtime protection. We believe that a special-purpose runtime instrumentation technique that would just keep track of tainted strings should reduce the runtime overhead even further.

2.1 Framework Overview

We start our discussion by focusing on the SQL injection example in Section 1.1.1. Conceptually, a vulnerability occurs because there is uninterrupted flow between a tainted object (as exemplified by String `userName` on line 3 in Figure **??**) and a sink (`execute` on line 5). It is important to point out that in Java every string is a separate object. Moreover, a `String` object is immutable, meaning that once it becomes tainted, it will always remain so. A vulnerability trace is a sequence of objects, such that every object is derived from the previous one, leading to a sink. Notice that the objects involved in a vulnerability trace are strings, represented in Java by standard library types `String`, `StringBuffer`, `StringBuilder`, `StringTokenizer`, etc. declared in package `java.lang`.

The overall goal of both static and runtime analyses is to locate such traces. While the example in Section 1.1.1 is quite simple, the trace is in fact 3 objects long:

1. The original source `java.lang.String` object on line 3;
2. The `java.lang.StringBuffer` object constructed when the Java compiler converts string concatenation into calls to `java.lang.StringBuffer.append`(...)[1];
3. The `java.lang.String` object that is the result of calling `StringBuffer.toString`() on the previous `StringBuffer` object.

[1]More recent versions of the Java starting with version 1.5 use the `StringBuilder` class, which offers an interface very similar to that of `StringBuffer`. The advantage of `StringBuilder` is that it is not `synchronized`, resulting in faster code.

Of course, large programs produce traces that are considerably longer and traces of length 20 and above are not uncommon. The longer a trace is, the more difficult it generally is to detect through code review or shallow analysis. Our techniques have been developed to find all traces, independent of their length. In the rest of this section we formalize the notions discussed above.

2.1.1 Tainted Object Propagation Problem

In this section we formalize the tainted object propagation problem first described in Section 1.1. We start by defining the terminology that was first informally introduced in Example 1.

Definition 2.1.1 An *access path* as a sequence of field accesses, array index operations, or method calls separated by dots. We denote the empty access path by ϵ; array indexing operations are indicated by [ ].
For instance, the result of applying access path `f.g` to variable `v` is `v.f.g`.

Definition 2.1.2 A *tainted object propagation problem* consists of a set of *source descriptors*, *sink descriptors*, *derivation descriptors*, and *sanitization descriptors*, as described below:

- *Source descriptors* of the form $\langle m, n, p\rangle$ specify ways in which user-provided data can enter the program. They consist of a source method m, parameter number n and an access path p to be applied to argument n to obtain the user-provided input. We use argument number -1 to denote the return result of a method call.

- *Sink descriptors* of the form $\langle m, n, p\rangle$ specify unsafe ways in which data may be used in the program. They consist of a sink method m, argument number n, and an access path p applied to that argument.

- *Derivation descriptors* of the form $\langle m, n_s, p_s, n_d, p_d\rangle$ specify how data propagates between objects in the program. They consist of a derivation method m, a source object given by argument number n_s and access path p_s, and a destination object given by argument number n_d and access path p_d. This derivation descriptor specifies that at a call to method m, the object obtained by applying p_d to argument n_d is derived from the object obtained by applying p_s to argument n_s.

- *Sanitization descriptors* of the form $\langle m, n_d, p_d \rangle$ specify *sanitization* methods that stop the propagation of taint between objects in the program. They consist of a derivation method m, a destination object given by argument number n_d and access path p_d. This sanitization descriptor specifies that at a call to method m, the object obtained by applying p_d to argument n_d is *not* tainted.

These descriptors formally specify how source methods in the program can generate tainted input and how sink methods can be exploited if unsafe input is passed to them. They also specify how string data can propagate between objects in the program by using string manipulation routines and when the flow of taint terminates.

A tainted object propagation problem is instantiated for any particular vulnerability type, such as SQL injections caused by parameter manipulation. Moreover, parts of the problem are application-specific. For instance, it is common to have application-specific sanitizers, whereas derivation routines are typically shared among most Java applications. Fortunately, the lists of sources and sinks are specific to the J2EE framework we use and can therefore be shared among all applications using those APIs. The issue of specification completeness is further discussed in Section 2.1.4.

2.1.2 Derivation and Sanitization Descriptors

While the notion of sources and sinks is intuitively clear, the subject of derivation and sanitization descriptors requires further discussion. In the absence of derived objects, to detect potential vulnerabilities we only need to know if a source object is used at a sink. Derivation descriptors are introduced to handle the semantics of strings in Java.

Because `String`s are immutable Java objects, string manipulation routines such as concatenation create brand new `String` objects, whose contents are based on the original `String` objects. Derivation descriptors are used to specify the behavior of string manipulation routines, so that taint can be explicitly passed among the `String` objects.

Unfortunately, there are numerous ways to obtain tainted objects from string objects in Java. Data contained in a string object propagates to any object derived from the string through string concatenation, substring extraction, and other similar routines. For instance, `s.toLowerCase()` is derived from string `s`. Similarly, the result of `s` + ";" is derived from string `s`. Finally,

```
String tainted = ...;
char[] chars = tainted.getChars();
for(int i = 0; i < chars.length; i++){
    char ch = chars[i];
    buf.append(ch);
}
String str = buf.toString();
con.executeQuery(str);
```

Figure 2.2: Character-level string manipulation not captured by our model.

`newStringTokenizer(s)` is derived from `s`, because the `StringTokenizer` object constructed out of a tainted string will produces potentially tainted tokens.

Most Java programs use built-in `String` libraries and can share the same set of derivation descriptors as a result. However, some Web applications use multiple `String` encodings such as Unicode, UTF-8, and URL encoding. If encoding and decoding routines propagate taint and are implemented using native method calls or character-level string manipulation, they also need to be specified as derivation descriptors. Sanitization routines that validate user input are also often implemented using character-level string manipulation.

It is possible to obviate the need for manual specification of derivation and sanitization descriptors with a static analysis that determines the relationship between strings passed into and returned by low-level string manipulation routines. We describe such an analysis in Section 2.1.4. However, such an analysis must be performed not just on the Java bytecode but on all the relevant native methods as well.

It is important to point out that the notion of derivation and sanitization descriptors we use is restricted to methods. We are unable to capture the creation of one string from characters of another if it does not involve a method call, as shown in Figure 2.2.

Example 2.1. We can formulate the problem of detecting parameter manipulation attacks that result in a SQL injection as follows: the source descriptor for obtaining parameters from an HTTP request is:

$$\langle \texttt{HttpServletRequest.getParameter(String)}, -1, \rangle,$$

where ϵ stands for the empty access path. A sink descriptor for SQL query execution is:

$$\langle \texttt{Connection.executeQuery(String)}, 1, \epsilon \rangle.$$

To allow the use of string concatenation in the construction of query strings, we use derivation descriptors:

$$\begin{array}{ll}\langle \texttt{StringBuffer.append(String)}, & 1, \epsilon, -1, \epsilon \rangle, \text{and} \\ \langle \texttt{StringBuffer.toString()}, & 0, \epsilon, -1, \epsilon \rangle\end{array}$$

Finally, in this example, we leave the list of sanitization descriptors empty. □

2.1.3 Security Violations

Below we formally define a security violation:

Definition 2.1.3 A *source object* for a source descriptor $\langle m, n, p \rangle$ is an object obtained by applying access path p to argument n of a call to m.

Definition 2.1.4 A *sink object* for a sink descriptor $\langle m, n, p \rangle$ is an object obtained by applying access path p to argument n of a call to method m.

Definition 2.1.5 Object o_2 is *derived* from object o_1, written *derived*(o_1, o_2), based on a derivation descriptor $\langle m, n_s, p_s, n_d, p_d \rangle$, if o_1 is obtained by applying p_s to argument n_s and o_2 is obtained by applying p_d to argument n_d at a call to method m.

Definition 2.1.6 An object is *tainted* if it is obtained by applying relation *derived* to a source object zero or more times.

Definition 2.1.7 A *security violation* occurs if a sink object is tainted. A security violation consists of a sequence of objects $o_1 \ldots o_k$ such that o_1 is a source object and o_k is a sink object and each object is derived from the previous one:

$$\underset{0 \le i < k}{\forall} i : \textit{derived}(o_i, o_{i+1}).$$

We refer to object pair $\langle o_1, o_k \rangle$ as a *source-sink pair*. When talking about vulnerability counts we will actually refer to the number of source-sink pairs our analysis detects.

2.1.4 Specifications Completeness

If a specification is incomplete, important errors will be missed even if we use a sound analysis that finds all vulnerabilities matching a specification. Therefore, the problem of obtaining a complete specification for a tainted object propagation problem is an important one. However, it is hardly a unique issue for program analysis, as many other projects require a specification to be provided [AE02, HCXE02, WFBA00].

To come up with a list of source and sink descriptors for vulnerabilities in our experiments, we used the documentation of the relevant J2EE library APIs. Since it is relatively easy to miss relevant descriptors in the specification, we used several techniques to make our problem specification *more* complete. For example, to find some of the missing source methods, we instrumented the Web applications to find places where application code is called by the application server.

We also used a static analysis to identify tainted objects that have no other objects derived from them, and examined methods into which these objects are passed. In our experience, some of these methods turned out to be obscure derivation and sink methods missing from our initial specification, which we subsequently added. However, despite our best efforts, we cannot claim specification completeness.

An interesting feature of our analysis framework is that it is generally not necessary to include character-level sanitization routines in the specification. This is because the analysis will be unable to follow the flow from the parameters of such routines to their return values, achieving the desired effect. It is, however, not acceptable to omit derivation routines, as this would miss some legitimate data flow through the program and threaten the soundness of our results.

2.2 Specifying Vulnerabilities in PQL

While a useful formalism, source, sink, derivation, and sanitization descriptors as defined in Section 2.1.1 are not a user-friendly way to describe security vulnerabilities. In both the static and dynamic analysis arenas, we have seen the development of various analysis specification techniques.

For example, for static analysis, questions about static program properties may be expressed as Datalog queries [WACL05] or type inference rules [KA05]. Datalog exposes the program intermediate representation (IR) as a set of relations. To determine static program properties, the user can subsequently query these relations. While giving the user complete control, Datalog queries expose too much of the program's internal representation to be practical for the casual use who does not want to learn the intricacies of the IR. The same argument applies to requiring the user to write runtime instrumentation code, leading to the development of numerous aspect-oriented systems such as AspectJ, etc. that make common tasks easier to accomplish [ea, KHH+01].

Our approach is to use PQL, a program query language. PQL is a general query language capable of expressing a variety of questions about program execution. A PQL query is a pattern describing a sequence of dynamic events that involves variables referring to *dynamic object instances*. Matching object instances are returned as the answer to the PQL query. PQL queries can be answered either statically or dynamically. In the static case, a conservative approximation of the answer is used: false positive matches may be introduced.

To make them accessible to developers, PQL queries are written in a familiar Java-like syntax. PQL serves as a layer of abstraction and, as a result, the user is not required to become familiar with the details of static program internal representation or the internals of an instrumentation framework.

In this report, we only use a relatively limited and stylized form of PQL queries to formulate tainted object propagation problems; a more extensive description of PQL is found elsewhere [MLL05]. Translation of tainted object propagation queries from PQL into static checkers and runtime instrumentation is described in more detail in Chapters ?? and 3, respectively.

2.2.1 Simple SQL Injection Query

Example 2.2. Figure 2.3 shows a PQL query for the SQL injection vulnerability in Example 1. It is important to point out and this is a relatively simple query example given here for the purpose of illustration that only addresses a small subset of all SQL injections that includes the code snippet in Figure ??. Queries capturing a wider range of vulnerabilities are discussed in Section 2.2.2.

Query `simpleSQLInjection` is described in more detail below. The **uses** clause of a PQL query declares all objects used in the query. The **matches**

```
query simpleSQLInjection()
returns
    object String param, derived;
uses
    object HttpServletRequest  req;
    object Connection          con;
    object StringBuffer        temp;
matches {
    param   = req.getParameter(_);

    temp.append(param);
    derived = temp.toString();

    con.execute(derived);
}
```

Figure 2.3: The PQL query for finding simple SQL injections.

clause specifies the sequence of events that must occur for a match to be found. Semicolons are used in PQL queries to indicate a sequence of events. The wildcard character _ is used instead of a variable name if the identity of the object to be matched is irrelevant. Finally, the **return** clause specifies source-sink pairs ⟨`param`, `derived`⟩ returned by the query. The **matches** clause is interpreted as follows:

1. object `param` must be obtained by calling `HttpServletRequest.getParameter`;
2. method `StringBuffer.append` must be called on object `temp` with `param` as the first argument;
3. method `StringBuffer.toString` must be called on `temp` to obtain object `derived`, and
4. method `execute` must be called with object `derived` passed in as the first parameter.

These operations must be performed in order; however, the invocations need *not* be consecutive and may be scattered across different methods. Query `simpleSQLInjection` matches the code in Example 1 with query variables `param` and `derived` matching the objects in `userName` and `query`. Query variable `temp` corresponds to the temporary `StringBuffer` created by the Java compiler for the string concatenation operation in Example 1. □

```
query main()
returns
    object Object sourceObj, sinkObj;
matches {
    sourceObj := source();
    sinkObj   := derived*(sourceObj);
    sinkObj   := sink();
}
```

Figure 2.4: Main query for finding source-sink pairs.

2.2.2 Queries for a Taint Propagation Problem

In this section we describe how generic tainted object propagation queries are formulated. There is a direct correspondence between source, sink, derivation, and sanitization descriptors used in the problem (definition 2.1.2) and parts of the PQL query shown in Figure 2.4.

Generic Taint Propagation Queries

Query `main` shown in Figure 2.4 computes source-sink object pairs corresponding to static or runtime security violations for a given tainted object propagation problem. Intuitively, query `main` matches pairs of objects, such that the first object comes from a source, the second goes into a sink, and the second object is derived from the first one using zero or more derivation steps. The source and sink objects are denoted in the query as `sourceObj` and `sinkObj`, respectively. Events separated by semicolons in query `main` must occur in order, but can be separated by other events (such as method calls, etc.).

Query `main` uses auxiliary subqueries `source`, `sink`, and `derived*` to constraint `sourceObj` and `sinkObj` values. Object `sourceObj` in `main` is returned by subquery `source`. Object `sinkObj` is the result of subquery `derived`* with `sourceObj` used as a subquery parameter and is also the result of subquery `sink`. Therefore, `sinkObj` returned by query `main` matches all tainted objects that are also sink objects.

Subquery `derived*` shown in Figure 2.5 defines a transitive derived relation: object `y` is transitively derived from object `x` by applying subquery `derived` zero or more times. This query takes advantage of PQL's subquery mechanism to define a transitive closure recursively.

```
query derived*(object Object x)
returns
    object Object y;
uses
    object Object temp;
matches {
    !sanitizer1(x); !sanitizer2(x); ...
    y    := x |
    temp := derived(x); y := derived*(temp);
}
```

Figure 2.5: Transitive derived relation `derived`*.

Instantiating Taint Propagation Queries

Subqueries `source`, `sink`, and `derived` used in `main` and `derived`* are specific to a particular tainted object propagation problem, as shown in the example below.

Example 2.3. This example describes subqueries `source`, `sink`, and `derived` shown in Figure 2.6 that can be used to match SQL injections, such as the one described in Example 1. Usually these subqueries are structured as a series of alternatives separated by |. The wildcard character _ is used instead of a variable name if the identity of the object to be matched is irrelevant.

Query `source` is structured as an alternation: `sourceObj` can be returned from a call to `req.getParameter` or `req.getHeader` for an object `req` of type `HttpServletRequest`; `sourceObj` may also be obtained by indexing into an array returned by a call to `req.getParameterValues`, etc. Query `sink` defines sink objects used as parameters of sink methods such as `java.sql.Connection.executeQuery`, etc. Query `derived` determines when data propagates from object `x` to object `y`. It consists of the ways in which Java strings can be derived from one another, including string concatenation, substring computation, etc. □

As can be seen from this example, subqueries `source`, `sink`, and `derived` map to source, sink, and derivation descriptors for the tainted object propagation problem. However, instead of descriptor notation for method parameters and return values, natural Java-like method invocation syntax is used.

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

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

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

Figure 2.6: PQL subqueries for finding SQL injections.

```
queries       ⟶  query*

query         ⟶  query qid ( [decl [, decl]*] )
                 [returns declList ; ]
                 [uses declList ; ]
                 [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 )
unifyStmt     ⟶  id := id
                 ( [idList] ) := qid ( idList )
seqStmt       ⟶  ( commaStmt ; )*

commaStmt     ⟶  altStmt ( , altStmt )*

altStmt       ⟶  stmt ( "|" stmt )*

typeName      ⟶  id ( . id )*

idList        ⟶  [ id ( , id )* ]

fieldAccess   ⟶  id . id

methodName    ⟶  typeName . id

qid           ⟶  [A-Za-z_][0-9A-Za-z_ ]*

qid           ⟶  [A-Za-z_][0-9A-Za-z_ ]*

namePattern   ⟶  [A-Za-z*_ ][0-9A-Za-z*_ ]*
```

Figure 2.7: BNF grammar specification for PQL.

Chapter 3

Runtime Analysis in SecuriFly

3.1 Matching PQL Queries at Runtime

PQL provides generic machinery for matching queries at runtime as described in the rest of this section. PQL queries are translated into non-deterministic finite-state automata (NFAs). The underlying application is instrumented so that all events relevant to the query being matched are recorded. When the application is executed, NFAs constructed on the bases of the PQL query run alongside the application collecting information about relevant program events.

Whenever the NFA corresponding to the `main` query enters an accept state, one of several outcomes can occur. If the **replaces** clause is present, another event is substituted in place of the event being replaced. This is especially useful for recovery, so that a safe action replaces a potentially unsafe one, as described in Section 3.4. If the **executes** clause is present, the code within the clause will be executed, which is useful for reporting vulnerabilities or terminating the application.

Finding dynamic matches to PQL queries involves the following steps:

Query translation. Translate each subquery into an NFA which takes an input event sequence, finds subsequences that match automaton, and reports the values bound to all returned query variables for each match.

Program instrumentation. Instrument the target application to record events relevant to the query being matched.

Query matching. Use a query matcher to interpret all the state machines over the execution trace collected as the program runs to find all matches.

Each of these steps is described in detail in Sections 3.1.1 — 3.1.3.

3.1.1 Translation From Queries To State Machines

A state machine representing a PQL query is composed of the following components:

- a set of states, which includes a start state, a fail state, and an accept state;
- a set of state transitions which may or may not be predicated;
- and a set of query variables taken from the original PQL query.

A *partial query match* is given by a current state and a set of *bindings* — mappings from variables in a PQL query to objects in the heap at runtime. A state transition specifies the event for 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.

Special Transitions

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

Skip transitions. A query specifies a *sub-sequence* of events to match. Unless noted otherwise with an exclusion statement, an arbitrary number of events of any kind are allowed in between consecutive matched statements. We represent this notion with a *skip transition*, which connects a state back to itself on any event that does not match the set of excluded events. Note that the accept state does not have a skip transition, so matches are reported only once.

ϵ **transitions**. An 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 original PQL queries to ease the translation process. 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 [ASU86]. 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.

Transitions Corresponding to Primitive Statements

We now present a syntax-directed approach to constructing the state machine for a query. The reader is encouraged to refer to the PQL grammar in Figure 2.7 as we describe how different primitive statements are translated. Before we can proceed, some additional notation is required.

Associated with each statement s in the query are two states, denoted $\mathit{bef}(s)$ and $\mathit{aft}(s)$, to refer to the states just before and after s is matched. For a query with statement s in the **matches** clause, the start and accept states of the query are states $\mathit{bef}(s)$ and $\mathit{aft}(s)$, respectively.

Definition 3.1.1 An attribute in event e with value x is *unifiable* with query statement s and the current set of bindings b if

- it refers to a query variable v that is unbound in b or bound in b to value x;
- or if the corresponding attribute in s has a literal constant value x.

Below we describe how the different PQL primitives are translated into NFAs.

Array and field operations. These are the primitive statements that correspond to single events during the execution. For a primitive statement s of type t, the transition from $\mathit{bef}(s)$ to $\mathit{aft}(s)$ is predicated by getting an input event e also of type t and that the attributes in e must be unifiable with those in statement s and the current bindings. 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 $\sim s'$, $\mathit{bef}(s) = \mathit{aft}(s)$. The default skip transition is modified to be predicated upon not matching s'.

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

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

Method invocation and creation points. 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 $\mathit{bef}(s)$ to t that matches the call, and bind v to the ID of the event. We create another transition from t to $\mathit{aft}(s)$ that matches a return with ID v. The skip transition from t back to itself is modified to exclude the match of the return event. Calls and returns are unified in a manner analogous to array and field operations. Object creation is handled in Java by invoking the method "`<init>`", and is translated into NFAs like any other method invocation.

Unification statements. A unification statement denoted by *unifyStmt* in Figure 2.7 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.

3.1.2 Instrumenting the Program

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 sent to the query matcher. The matcher updates the state of all pending matches and then returns control to the application. For instance, the NFA that corresponds to a PQL query that concerns calls to method `StringBuffer.toString()` will be notified each time this method is invoked. Moreover, the value of the `this` parameter will be passed to the NFA also.

The matcher 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.

3.1.3 The Runtime Query Matcher

The matcher 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 currently processed event produces a new possible partial match where that transition is taken.

Handling Non-Determinism

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.

Handling Subqueries

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.

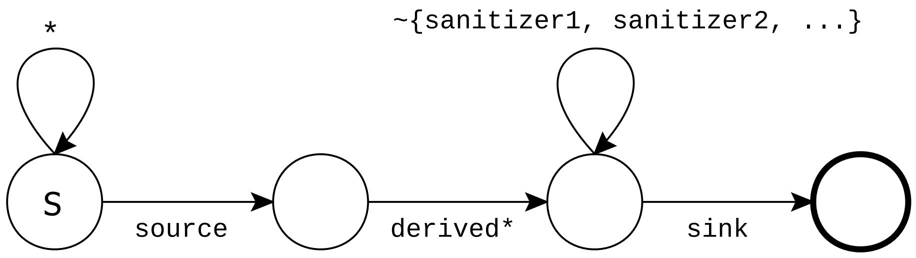

Figure 3.1: State machine that corresponds to the `main` PQL query.

Handling Accept States

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.

3.2 Translating Vulnerability Queries

The previous section presented a generic procedure for translating from PQL queries to NFAs. This section discusses the state machines that are created for the specific vulnerability queries shown in Figures 2.4 — 2.6. For all the NFAs discussed in this section, `S` marks the start state and thick-edged graph nodes are accept states. For edges, $*$ marks an edge that can be taken on any input. Exclusion notation $\sim \texttt{e}_1, \texttt{e}_2, \ldots$ on graph edges marks an edge that can be taken on any input events *except* $e_1, e_2, \ldots$.

Query `main`. The NFA in Figure 3.1 for the `main` PQL query consists of invocations of subqueries `source`, `sink`, and `derived`$*$. This corresponds to a piece of data that is read from a source, derived from using zero or more steps, and then falls into a sink. This exactly matches the notion of a tainted object propagation problem in Section 2.1.1.

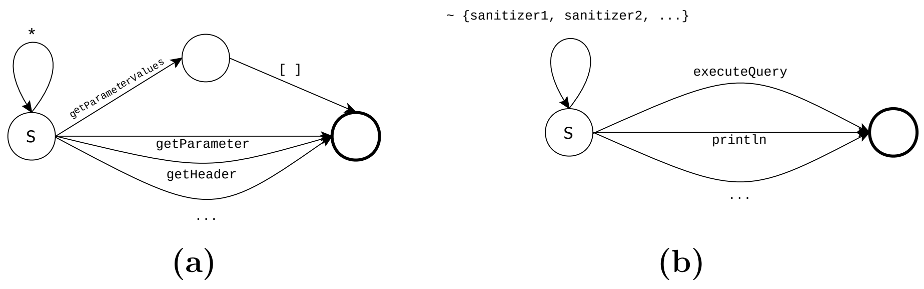

Figure 3.2: State machines corresponding to the (a) `source` and (b) `sink` PQL queries.

It is important to point out that the transition on the `sink` edge leading to the accepting node is only allowed when no sanitizer calls are encountered (sanitizers are denoted by `sanitizer1`, `sanitizer2`, etc.). This is important since it is possible for `derived∗` query to complete without encountering a sanitizer. Once the `derived∗` step finishes, a sanitizer could be applied to the same object as the one passed into a sink.

Query `source`. The `source` NFA shown in Figure 3.2(a) accepts on methods calls to source methods such as `getParameter`, etc. One complication is the treatment of return values of a call to `getParameterValues`. It is required that the returned array be indexed, as represented by the edge marked with "[ ]" for the state machine to accept. A similar technique is used to make values of a map returned from `getParameterMap` tainted, except that several possibilities exist: method `get` needs to be called on the map returned from the call; alternatively, an iterator could be constructed over the map values by calling `values`().`iterator`() and then method `next`() could be called on the iterator.

Queries `sink` and `derived`. Queries `sink` and `derived` consist of an alternation of methods that correspond to sink and derivation descriptors, respectively. Notice that the `sink` and `derived` NFAs in shown in Figures 3.2(b) and 3.3(a) only accepts if no sanitizer is encountered.

Query `derived∗`. The NFA in Figure 3.3(b) is self-recursive and corresponds to zero or more invocations of subquery `derived`. When the `temp` node is reached, a new state machine is created to interpret the recursive invocation of `derived∗`. Eventually, the top branch from the start node will be taken, thus completing the subquery match.

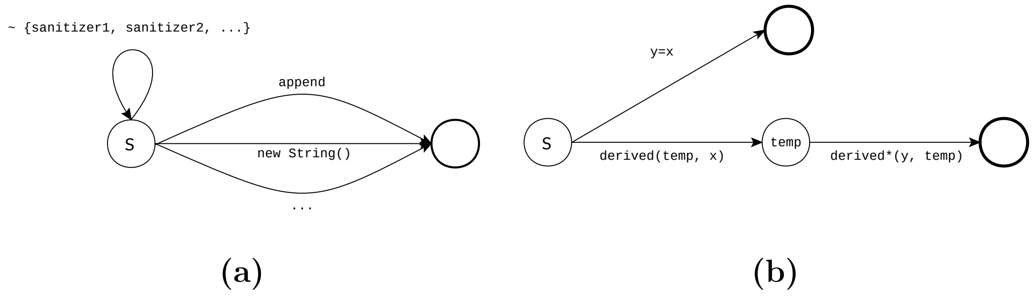

Figure 3.3: State machines corresponding to the (a) `derived` and (b) `derived`∗ PQL queries.

3.3 Reducing Instrumentation Overhead

Instrumentation code is inserted only at those program points that might generate an event of interest for the specific query. To reduce the number of instrumentation points, a simple type analysis excludes operations on types not related to objects in the query. However, this is often not enough. For example, in the case of query `derived`, most `String` and `StringBuffer` operations would have to be instrumented. Since there are many such method calls, this results in a high overhead.

In order to reduce the overhead further, we use the results of our static analysis, further described in Martin et al. [MLL05], to reduce the instrumentation by excluding statements that cannot refer to objects involved in any match of the query. For queries capturing the tainted object problem, we only need to instrument calls on a path from a source to a sink, which account for a small portion of all string-related method calls. Also, as described in Martin et al. , instead of collecting full execution traces and post-processing them, our system tracks all the partial matches as the program executes and takes action immediately upon recognizing a match.

While the overhead reduction achieved with static analysis is very significant, we believe that even greater improvements can be made with special-purpose instrumentation that tracks the flow of taint in a way that is conceptually similar to runtime tainting in Perl [WCS96]. While not as flexible as our PQL-based approach, a lookup table kept on the side at runtime that records the taint status of every `String`, `StringBuffer`, and `StringBuilder` object would go a long way towards improving Web application security. However, at the same time, this simple representation would make the no-

```
query main()
returns
    object Object sourceObj, sinkObj;
matches {
    sourceObj := source();
    sinkObj   := derived*(sourceObj);
    sinkObj   := sink();
}
replaces java.sql.PreparedStatement.prepareStatement(sink)
    with SQL.SafePrepare(sourceObj, sinkObj);
replaces java.sql.Statement.executeQuery(sink)
    with SQL.SafeExecute(sourceObj, sinkObj);
...
```

Figure 3.4: Augmented `main` query for recovering from exploits at runtime.

tion of a map, whose values are tainted hard to model.

3.4 Dynamic Recovery from Vulnerabilities

Figure 3.4 presents an augmented version of query `main` that has recovery capabilities. As can be seen from the augmented query, each operation that can unsafely use tainted data receives a `replaces` clause in the augmented `main` query.

When a possibly relevant sink is reached, any matches that have completed and which are consistent with the event being replaced are gathered, and if such matches are present, the replacing method is executed instead. Since every argument to the `replaces` clause except `sourceObj` appears in the replaced event, `sourceObj` is the only variable that may have multiple values. The replacement method provides a safe alternative for each of the sinks in the query. In general, the replacement method sanitizes tainted values. The kind of sanitization applied is different depending on the type of vulnerability and also the method that is being replaced.

3.4.1 Built-in Sanitization

While it is generally up to the user to provide the proper sanitization routines, in the case of SQL and HTML, PQL provides a library of simple and generic sanitization functions that can be used if application-specific sanitizers are unknown.

For example, sanitization methods `SafePrepare` and `SafeExecute` work by finding all substrings within string `sinkObj` that match any of the possible values for string `sourceObj`. A new SQL query string is constructed with all SQL metacharacters in any such substring quoted. This new query is then passed to `prepareStatement` or `executeQuery`, respectively.

Example 3.1. Consider a `sourceObj` that refers to string `'O'Brian'`. Suppose `sinkObj` refers to string

```
SELECT * FROM Users WHERE name = 'O'Brian'
```

The result of applying `SafePrepare` will be

```
SELECT * FROM Users WHERE name = 'O''Brian'
```

which escapes the string within the quotation marks. In the MySQL dialect of SQL, this escaping is achieved by doubling quotation marks. □

Using this relatively simple escaping technique we were able to defend against two SQL injections in two of our benchmark programs, `webgoat` and two more in `road2hibernate` for which we had derived effective attacks.

3.4.2 Shortcomings of Built-in Sanitizers

However, in general, this escaping mechanism is quite simplistic and may not always result in the desirable output. For example, if `sinkObj` uses the upper-case version of `sourceObj`, it will not be matched. Similarly, the `hibernate` object persistence library performs heavy processing on user input, but fails to actually quote the dangerous components of it verbatim. The following input

```
bob' or 1=1
```

will be converted by `hibernate` into

```
bob' or '1'='1'
```

Because of this existing quoting mechanism, which actually does nothing to protect against SQL injections, it was necessary to modify the query to perform the substitution step at the interface *between* `road2hibernate` and `hibernate`, an open-source object-persistence library, rather than between the `hibernate` and the database itself.

This illustrates a more general point about applying sanitization: where it needs to be placed is often open for discussion. While our approach of applying it right before the sink works in most cases, it is not necessarily most efficient. In many cases, the proper place to insert sanitization — both in the code and at runtime — is between abstraction boundaries or before a piece of data is places into a data structure, etc.

Chapter 4

Experimental Results

Our first test of the runtime system consisted of running exploits that we created based on statically found vulnerabilities in SecuriBench applications. Our exploits focused on SQL injection and cross-site scripting attacks, as these are the easiest to mount and the results are most apparent. All of these exploits were detected and thwarted when runtime recovery was enabled.

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 PQL query is implemented as five separate state machines, one for each query. The effect of the instrumentation is to track all `String`s that either are directly user-controlled or that are derived from it, and to report a match if such a user-controlled string falls unsafely into Java's SQL interface.

Note that even if a *given user input* is harmless in a particular execution, the data will still flow the same way, and thus will still be matched. 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 may not be suitable outside of a debugging context. Implementing a second level of checking that actually considers the values or just logging potentially malicious input as well as the injection paths may be appropriate. The rest of this section focuses on performance overhead incurred with different versions of our runtime instrumentation.

Benchmark	Instrumentation points		Runtime			Overhead	
	U	O	Uninstrumented	U	O	U	O
`webgoat`	604	69	.024	.054	.033	125%	37%
`personalblog`	3,209	36	.040	.069	.049	72%	22%
`road2hibernate`	4,146	779	2.224	2.443	2.362	9%	3%
`snipsnap`	3,305	542	.073	.096	.080	31%	9%
`roller`	2,960	96	.008	.012	.008	50%	< 1%

Figure 4.1: Summary of the number of instrumentation points, running times, dynamic overhead, both with and without optimizations. “U” and “O” stand for *unoptimized* and *optimized* runtime instrumentations, respectively. All times are given in seconds.

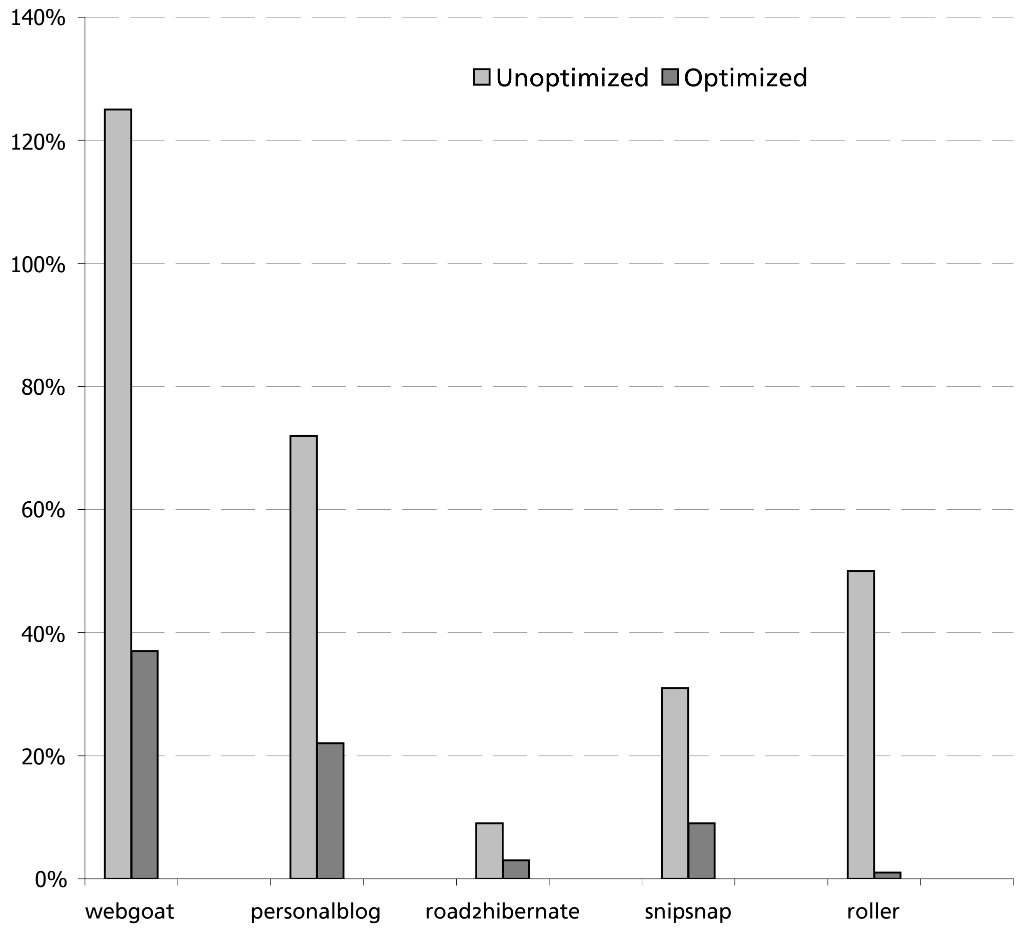

Figure 4.2: Runtime analysis overhead comparison.

4.1 Performance Summary

Figure 4.1 summarized the runtime analysis overhead. Results are presented for both the unoptimized ("U") and the optimized ("O") runtime analysis versions. Several SecuriBench applications are missing from the table, as we were unable to install them for runtime analysis due to complex configuration and database dependency issues. Columns 2 and 3 show the number of instrumentation points that were inserted by the runtime instrumentation described in Chapter 3.

Columns 4 — 6 summarize the running times measured in seconds. Measuring Web application running times presents a number of unique challenges not present in command-line applications. The times we report for the Web applications reflect the average amount of time required to serve a *single* page in response to a single HTTP request, as measured by the standard profiling tool JMeter [Fou]. The only exception is `road2hibernate`, which is a command-line program and its time is a simple start-to-finish timing. Finally, columns 7 and 8 summarize the overhead with the unoptimized and optimized versions of the analysis.

Overall, 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 over tens of thousands of classes.

4.2 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 would 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 provably not present on any path from user input to database access. Exploiting static information dramatically reduces both the number of instrumentation points and the overhead of the system, as shown in Figure 4.1. Figure 4.2 presents a graphical summary of runtime overhead results.

The reduction in the number of instrumentation points due to static optimization can be as high as 97% in `roller` and 99% in `personalblog`. Reductions in the number of instrumentation points result in dramatically smaller overheads. For instance, in `webgoat`, the overhead was cut almost in half in the optimized version.

Chapter 5

Related Work

This section gives an overview of dynamic analysis techniques that address memory safety vulnerabilities prevalent in C and C++ programs as well as runtime techniques pertaining to Web application vulnerabilities.

5.1 Vulnerabilities in Type-Unsafe Languages

A range of compiler extensions discussed below has been used to protect against memory-based attacks prevalent in C programs such as format string violations and buffer overruns. A good overview of these techniques is given in Kc et al. [KEKK02].

FormatGuard, a compiler modification, injects code to dynamically check and reject all `printf`-like function calls where the number of arguments does not match the number of "%" specifiers in the format string [CBB+01]. Of course, only applications that are re-compiled using FormatGuard will benefit from its protection. Also, one technical shortcoming of FormatGuard is that it does not protect user-defined wrappers for the `printf` family of routines. An unfortunate consequence of the design choices of FormatGuard is that programs with format string vulnerabilities remain vulnerable to denial of service attacks.

A wide range of approaches focuses on runtime buffer overrun protection. Products such as StackGuard [CPM+98], StackShielf [Ano02] and the /`GS` switch implemented in the later version of the Microsoft Visual Studio compilers [Cor05] all use similar techniques to provide protection against stack smashing exploits. StackGuard works by placing a "canary" word next to the return address on the stack. If the canary word has been altered when the function returns, then a stack smashing attack has been attempted while within the function. The StackGuard-protection program responds by emitting an intruder alert and then halting the program. Unfortunately, while generally effective, this sort of stack protection can still be circumvented with more sophisticated attack techniques such as spoofing the canary, etc. [Ric02, BK00].

PointGuard focuses on heap-based buffer overrun exploits [CBJW03]. PointGuard-protected programs encrypts all pointers while they reside in memory and decrypts them only before they are loaded to a CPU register. Similarly to FormatGuard and StackGuard, PointGuard is implemented as an extension to the GCC compiler, which injects the necessary instructions at compilation time, allowing a pure-software implementation of the scheme. The overhead incurred with PointGuard may, however, be prohibitively expensive [TCV04].

Kiriansky et al. propose program shepherding, a policy-driven mechanism for closely monitoring and dynamically controlling the flow of program execution [KBA02]. The advantage of program shepherding is that the original program does not need to be recompiled. They define different default and customizable security policies for code based on the nature of its origin, whether it was loaded from the local file system, generated by the running program itself, or if it self-mutated. Their system is integrated into an interpreter, which enables the sandboxed checking of running applications and monitoring of their control-flow. While the functionality of this approach is attractive, the fact that it is interpreted makes for significant overhead.

5.2 Runtime Analysis for WebApp Security

Scott et al. present a structuring technique which helps designers abstract security policies from large Web applications [SS02]. Their system consists of a specialized Security Policy Description Language which is used to program an application-level firewall. Security policies are written and compiled for execution on the security gateway. The security gateway dynamically analyses and transforms HTTP requests and responses to enforce the specialized policy. To the best of our knowledge, this system has not been applied to large Web applications.

5.2.1 Protection from SQL Injections

Several techniques focus on SQL injections exclusively. Buehrer et al. propose a technique that is based on comparing, at execution time, the parse tree of the SQL statement *before* inclusion of user input with that resulting *after* the inclusion of user-provided input [BWS05]. SQLRand used SQL keyword randomization in order to create SQL language keywords that are not easily guessable by the attacker, thus foiling most SQL injection techniques that involve adding extra SQL commands [BK04].

AMNESIA is a model-based approach that detects illegal queries before they are executed on the database [HO05a, HO05b, HO06, HVO06]. In its static part, the technique uses program analysis to automatically build a model of the legitimate queries that could be generated by the application. In its dynamic part, this technique uses runtime monitoring to inspect the dynamically-generated queries and check them against the statically-built model. Depending on the quality of the statically-derived model, their technique may suffer from both false positives and false negatives. Moreover, it is unclear how their static analysis would scale to large programs, as it has only been evaluated with relatively small benchmarks.

5.2.2 Dynamic Taint Propagation

Dynamic taint propagation described in Haldar et al. borrows much from our runtime technique [HCF05]. In contrast to our technique, they use heuristics similar to those use in the Perl taint mode [WCS96] to determine which `String`s need to be untained at runtime. I.e. matching against regular expressions is assumed to be an untainting operation. However, unlike SecuriFly, their approach is unable to provide recovery from vulnerabilities.

Pietraszek et al. propose CSSE, a system that modifies the PHP interpreter to tag strings to distinguish those that are developer-supplied from those that are provided as input. Since CSSE tracks where the different segments of a string originate, it is able to provide user string escaping or recovery in a manner similar to that of our runtime technique. Su et al. describe SqlCheck, a similar system for SQL injection detection that works on both Java and PHP code [SW06]. SqlCheck has been shown effective at preventing SQL injections in a range of medium-sized Web applications.

PHPrevent is a project that focuses on securing PHP applications [NTGG+05]. While similar in spirit to our runtime protection described in Chapter 3, PHPrevent uses a modified PHP interpreter to precisely track taint at runtime. Unlike our approach, however, the granularity of taint tracking is greater: tainting is recorded and propagated at the level of individual characters. Their approach to untainting is to escape parts of the input contained in the output. However, their notion of white-listing the allowed input is somewhat arbitrary and will not necessarily work for applications such bulletin boards that require some of the HTML tags to pass through. This is not unlike our notion of built-in sanitizers discussed in Sections 3.4.1 and 3.4.2.

5.3 PQL and Runtime Matching Formalisms

In addition to PQL, other formalisms have been developed to talk about events that occur during program execution. We briefly summarize some of that work here.

5.3.1 Aspect-Oriented Formalisms

PQL attaches user-specified actions to subquery matches; this capability puts PQL in the class of *aspect-oriented* programming languages [KHH+01, OL01]. Maya [BH02] and AspectJ [KHH+01] attach actions based on syntactic properties of individual statements in the source code. The DJ system defines aspects as traversals over a graph representing the program structure [OL01].

PQL 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" 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 [MK03]. Walker and Veggers introduce the concept of *declarative event patterns*, in which regular expressions of traditional pointcuts are used to specify when advice should run [Wal00]. Allan et al. extend this further by permitting PQL-like free variables in the patterns [AAC+05]. PQL differs from these systems in that its matching machinery can recognize non-regular languages, and in exploiting advanced pointer analysis to prove points irrelevant to eventual matches.

5.3.2 Other Program Query Languages

Systems like ASTLOG [Cre97] and JQuery [JdV03] permit patterns to be matched against source code; Liu et al. [LRY^{+}04] extend this concept to include parametric pattern matching [Bak95]. 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 [LHS97]. 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.

The Partiqle system [GOA05] 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.

5.3.3 Analysis Generators

PQL follows in a tradition of powerful tools that take small specifications and use them to automatically generate analyses. Metal [HCXE02] and SLIC [BR02] 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 [LMRC05] 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 [NS03] also presents a general technique for dynamic analyses on binaries.

Bibliography

[AAC+05] Chris Allan, Pavel Augustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 345 – 364, October 2005.

[AE02] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the Symposium on Security and Privacy*, May 2002.

[Anl02a] Chris Anley. Advanced SQL injection in SQL Server applications. `http://www.nextgenss.com/papers/advanced_sql_injection.pdf`, 2002.

[Anl02b] Chris Anley. (more) advanced SQL injection. `http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf`, 2002.

[Ano02] Anonymous. StackShield. `http://www.angelfire.com/sk/stackshield`, 2002.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bak95] Brenda S. Baker. Parameterized pattern matching by Boyer-Moore type algorithms. In *Proceedings of the Symposium on Discrete Algorithms*, pages 541–550, January 1995.

[Bar03] Darrin Barrall. Automated cookie analysis. http://www.spidynamics.com/support/whitepapers/SPIcookies.pdf, 2003.

[BH02] Jason Baker and Wilson Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 86 – 95, March 2002.

[BK00] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack Magazine*, 0xa(0x38), May 2000.

[BK04] Stephen Boyd and Angelos D. Keromytis. SQLrand: preventing SQL injection attacks. In *Proceedings of the Applied Cryptography and Network Security Conference*, pages 292–304, June 2004.

[BR02] Thomas Ball and Sriram Rajamani. SLIC: a specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, January 2002.

[BWS05] Gregory T. Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the International Workshop on Software Engineering and Middleware*, pages 106–113, September 2005.

[CBB$^+$01] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: automatic protection from printf format string vulnerabilities. In *Proceedings of the Usenix Security Symposium*, pages 191–200, August 2001.

[CBJW03] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard^{TM}: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the Usenix Security Symposium*, August 2003.

[CGI] CGI Security. The cross-site scripting FAQ. http://www.cgisecurity.net/articles/xss-faq.shtml.

[Chi04] Chinotec Technologies. Paros—a tool for Web application security assessment. http://www.parosproxy.org, 2004.

[Coo03] Steven Cook. A Web developers guide to cross-site scripting. http://www.giac.org/practical/GSEC/Steve_Cook_GSEC.pdf, 2003.

[Cor05] Microsoft Corporation. Microsoft minimizes threat of buffer overruns, builds trustworthy applications. http://download.microsoft.com/documents/customerevidence/12374_Microsoft_GS_Switch_CS_final.doc, 2005.

[CPM+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the Usenix Security Conference*, pages 63–78, January 1998.

[Cre97] Roger F. Crew. ASTLOG: a language for examining abstract syntax trees. In *Proceedings of the Usenix Conference on Domain-Specific Languages*, pages 229–242, 1997 1997.

[ea] Bill Burke et. al. JBoss AOP. http://labs.jboss.com/portal/jbossaop/index.html.

[Fou] Apache Foundation. Apache JMeter. http://jakarta.apache.org/jmeter/.

[Fri04] Steve Friedl. SQL injection attacks by example. http://www.unixwiz.net/techtips/sql-injection.html, 2004.

[GOA05] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 385–402, October 2005.

[HCF05] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, December 2005.

[HCXE02] Seth Hallem, Ben Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 69–82, June 2002.

[HO05a] William G. J. Halfond and Alessandro Orso. AMNESIA: analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the International Conference on Automated Software Engineering*, pages 174–183, November 2005.

[HO05b] William G. J. Halfond and Alessandro Orso. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In *Proceedings of the International ICSE Workshop on Dynamic Analysis*, pages 22–28, May 2005.

[HO06] William G. J. Halfond and Alessandro Orso. Preventing SQL Injection Attacks Using AMNESIA. In *Proceedings of the International Conference on Software Engineering (formal demo track)*, May 2006.

[Hu04] Deyu Hu. Preventing cross-site scripting vulnerability. http://www.giac.org/practical/GSEC/Deyu_Hu_GSEC.pdf, 2004.

[HVO06] William G. J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the International Symposium on Secure Software Engineering*, March 2006.

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the Conference on World Wide Web*, pages 40–52, May 2004.

[JdV03] Doug Janzen and Kris de Volder. Navigating and querying code without getting lost. In *Proceedings of the Conference on Aspect-Oriented Software Development*, pages 178–187, March 2003.

[KA05] John Kodumal and Alex Aiken. Banshee: a scalable constraint-based analysis toolkit. In *Proceedings of the International Static Analysis Symposium*, September 2005.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the Usenix Security Symposium*, pages 191–206, August 2002.

[KEKK02] Gaurav S. Kc, Stephen A. Edwards, Gail E. Kaiser, and Angelos Keromytis. CASPER: compiler-assisted securing of programs at runtime. Technical Report CUCS-025-02, Columbia University, 2002.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[Kle02a] Amit Klein. Cross site scripting explained. `http://crypto.stanford.edu/cs155/CSS.pdf`, June 2002.

[Kle02b] Amit Klein. Hacking Web applications using cookie poisoning. `http://www.cgisecurity.com/lib/CookiePoisoningByline.pdf`, 2002.

[Kle04] Amit Klein. Divide and conquer: HTTP response splitting, Web cache poisoning attacks, and related topics. `http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf`, 2004.

[Kos04] Stephen Kost. An introduction to SQL injection attacks for Oracle developers. `http://www.net-security.org/dl/articles/IntegrigyIntrotoSQLInjectionAttacks.pdf`, 2004.

[Kra05] Michael Krax. Mozilla foundation security advisory 2005-38. `http://www.mozilla.org/security/announce/mfsa2005-38.html`, 2005.

[LHS97] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 304–317, October 1997.

[Lit03a] David Litchfield. Oracle multiple PL/SQL injection vulnerabilities. http://www.securityfocus.com/archive/1/385333/2004-12-20/2004-12-26/0, 2003.

[Lit03b] David Litchfield. *SQL Server Security*. McGraw-Hill Osborne Media, 2003.

[LMRC05] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 364–377, January 2005.

[LRY$^+$04] Yanhong A. Liu, Tom Rothamel, Fuxiang Yu, Scott D. Stoller, and Nanjun Hu. Parametric regular path queries. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 219–230, June 2004.

[MK03] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, pages 105–121, November 2003.

[MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors using PQL: a program query language. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2005.

[Net04a] NetContinuum, Inc. The 21 primary classes of Web application threats. https://www.netcontinuum.com/securityCentral/TopThreatTypes/index.cfm, 2004.

[Net04b] 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.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: a program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89, 2003.

[NTGG$^+$05] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening Web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, June 2005.

[OL01] Doug Orleans and Karl Lieberherr. DJ: dynamic adaptive programming in Java. In *Proceedings of Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[Oll04] Gunter Ollmann. Second-order code injection attacks. http://www.nextgenss.com/papers/SecondOrderCodeInjection.pdf, 2004.

[Ope04] Open Web Application Security Project. The ten most critical Web application security vulnerabilities. http://umn.dl.sourceforge.net/sourceforge/owasp/OWASPTopTen2004.pdf, 2004.

[Ope05] Open Web Application Security Project. A guide to building secure Web applications. http://easynews.dl.sourceforge.net/sourceforge/owasp/OWASPGuide2.0.1.pdf, 2005.

[Ric02] Gerardo Richarte. Bypassing the StackShield and StackGuard protection. http://www.coresecurity.com/files/files/11/StackguardPaper.pdf, April 2002.

[Spe02a] Kevin Spett. Cross-site scripting: are your Web applications vulnerable. http://www.spidynamics.com/support/whitepapers/SPIcross-sitescripting.pdf, 2002.

[Spe02b] Kevin Spett. SQL injection: are your Web applications vulnerable? http://downloads.securityfocus.com/library/SQLInjectionWhitePaper.pdf, 2002.

[SS02] David Scott and Richard Sharp. Abstracting application-level Web security. In *Proceedings of International World Wide Web Conference*, May 2002.

[SS04] Moran Surf and Amichai Shulman. How safe is it out there? http://www.imperva.com/download.asp?id=23, 2004.

[SW06] Zhendong Su and Gary Wassermann. The essence of command injection attacks in Web applications. *ACM SIGPLAN Notes*, 41(1):372–382, 2006.

[TCV04] Nathan Tuck, Brad Calder, and George Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *Proceedings of the International Symposium on Microarchitecture*, December 2004.

[WACL05] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog and binary decision diagrams for program analysis. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, November 2005.

[Wag05] Stefan Wagner. Towards software quality economics for defect-detection techniques. In *Proceedings of the Annual IEEE/NASA Software Engineering Workshop*, April 2005.

[Wal00] David Walker. A type system for expressive security policies. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 254–267, January 2000.

[WCS96] Larry Wall, Tom Christiansen, and Randal Schwartz. *Programming Perl*. O'Reilly and Associates, Sebastopol, CA, 1996.

[WFBA00] David Wagner, Jeff Foster, Eric Brewer, and Alex Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 3–17, February 2000.

[XA06] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the Usenix Security Symposium*, pages 271–286, August 2006.