Finding Security Vulnerabilities in Java Applications with Static Analysis

Benjamin Livshits and Monica S. Lam

Computer Science Department
Stanford University

{livshits, lam}@cs.stanford.edu

Technical Report

September 25, 2005

Contents

Abstract

This report proposes a static analysis technique for detecting many recently discovered application vulnerabilities such as SQL injections, cross-site scripting, and HTTP splitting attacks. These vulnerabilities stem from unchecked input, which is widely recognized as the most common source of security vulnerabilities in Web applications. We propose a static analysis approach based on a scalable and precise points-to analysis.

In our system, user-provided specifications of vulnerabilities are automatically translated into static analyzers. Our approach finds all vulnerabilities matching a specification in the statically analyzed code. Results of our static analysis are presented to the user for assessment in an auditing interface integrated within Eclipse, a popular Java development environment.

Our static analysis found 29 security vulnerabilities in nine large, popular open-source applications, with two of the vulnerabilities residing in widely-used Java libraries. In fact, all but one application in our benchmark suite had at least one vulnerability. Context sensitivity, combined with improved object naming, proved instrumental in keeping the number of false positives low. Our approach yielded very few false positives in our experiments: in fact, only one of our benchmarks suffered from false alarms.

This report is an extended version of the material that appears in [LL05].

SECTION 1

Introduction

The security of Web applications has become increasingly important in the last decade. More and more Web-based enterprise applications deal with sensitive financial and medical data, which, if compromised, can cause significant downtime and millions of dollars in damages. It is crucial to protect these applications from hacker attacks.

However, the current state of application security leaves much to be desired. The 2002 Computer Crime and Security Survey conducted by the Computer Security Institute and the FBI revealed that, on a yearly basis, over half of all databases experience at least one security breach and an average episode results in close to $4 million in losses [Com02]. The survey also noted that Web crime has become commonplace. Web crimes range from cyber-vandalism (e.g., Web site defacement) at the low end, to theft of sensitive information and financial fraud at the high end. A recent penetration testing study performed by the Imperva Application Defense Center included more than 250 Web applications from e-commerce, online banking, enterprise collaboration, and supply chain management sites [Web04]. Their vulnerability assessment concluded that at least 92% of Web applications are vulnerable to some form of hacker attacks. Security compliance of application vendors is especially important in light of recent U.S. industry regulations such as the Sarbanes-Oxley act pertaining to information security [Bea03, Gro04].

A great deal of attention has been given to network-level attacks such as port scanning, even though, about 75% of all attacks against Web servers target Web-based applications, according to a recent survey [Hul01]. It is easy to underestimate the potential level of risk associated with sensitive information within databases accessed through Web applications until a severe security breach actually occurs. Traditional defense strategies such as firewalls do not protect against Web application attacks, as these attacks rely solely on HTTP traffic, which is usually allowed to pass through firewalls unhindered. Thus, attackers typically have a direct line to Web applications.

Many projects in the past focused on guarding against problems caused by the unsafe nature of C, such as buffer overruns and format string vulnerabilities [CPM+98, STFW01, WFBA00]. However, in recent years, Java has emerged as the language of choice for building large complex Web-based systems, in part because of language safety features that disallow direct memory access and eliminate problems such as buffer overruns. Platforms such as J2EE (Java 2 Enterprise Edition) also promoted the adoption of Java as a language for implementing e-commerce applications such as Web stores, banking sites, etc.

A typical Web application accepts input from the user browser and interacts with a back-end database to serve user requests; J2EE libraries make these common tasks easy to code. However, despite Java language's safety,

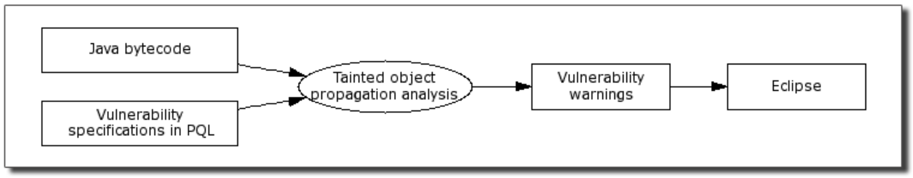

Figure 1: Architecture of our static analysis framework.

it is possible to make logical programming errors that lead to vulnerabilities such as SQL injections [Anl02a, Anl02b, Fri04] and cross-site scripting attacks [CGI, Hu04, Spe02a]. Discovered several years ago, these attack techniques are now commonly used to create exploits by malicious hackers. A score of recently discovered vulnerabilities can be attributed to these attacks []. A simple programming mistake can leave a Web application vulnerable to unauthorized data access, unauthorized updates or deletion of data, and application crashes leading to denial-of-service attacks.

1.1 Causes of Vulnerabilities

Of all vulnerabilities identified in Web applications, problems caused by *unchecked input* are recognized as being the most common [Ope04b]. 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" [Net04] and the "OWASP Secure Development Guide [Ope04a]".

1.2 Code Auditing for Security

Many attacks described in the previous section can be detected with code auditing. Code reviews pinpoint potential vulnerabilities before an application is run. In fact, most Web application development methodologies recommend a security assessment or review step as a separate development phase after testing and *before* application deployment [Ope04a, Ope04b].

Code reviews, while recognized as one of the most effective defense strategies [HL01], are time-consuming, costly, and are therefore performed infrequently. Security auditing requires security expertise that most developers do not possess, so security reviews are often carried out by external security consultants, thus adding to the cost. In addition to this, because new security errors are often introduced as old ones are corrected, *double-audits* (auditing the code twice) is highly recommended. The current situation calls for better tools that help developers avoid introducing vulnerabilities during the development cycle.

1.3 Static Analysis

We propose a tool based on a static analysis for finding vulnerabilities caused by unchecked input. Users of the tool can describe vulnerability patterns of interest succinctly in PQL [MLL05], which is an easy-to-use program query language with a Java-like syntax. Our tool, as shown in Figure 1, applies user-specified queries to Java bytecode and finds all potential matches statically. The results of the analysis are integrated into Eclipse, a popular open-source Java development environment [DFK+04], making the potential vulnerabilities easy to examine and fix as part of the development process.

The advantage of static analysis is that it can find all potential security violations without executing the application. The use of bytecode-level analysis obviates the need for the source code to be accessible. This is especially important since libraries whose source is unavailable are used extensively in Java applications. Our approach can be applied to other forms of bytecode such as MSIL, thereby enabling the analysis of C# code [MRM03].

Our tool is distinctive in that it is based on a precise context-sensitive pointer analysis that has been shown to scale to large applications [WL04]. This combination of scalability and precision enables our analysis to find all vulnerabilities matching a specification within the portion of the code that is analyzed statically. In contrast, previous practical tools are typically unsound [BPS00, HCXE02]. Without a precise analysis, these tools would find too many potential errors, so they only report a subset of errors that are likely to be real problems. As a result, they can miss important vulnerabilities in programs.

1.4 Contributions

This report makes the following contributions.

A unified analysis framework. We unify multiple, seemingly diverse, recently discovered categories of security vulnerabilities in Web applications and propose an extensible tool for detecting these vulnerabilities using a sound yet practical static analysis for Java.

A powerful static analysis. Our tool is the first practical static security analysis that utilizes fully context-sensitive pointer analysis results. We improve the state of the art in pointer analysis by improving the object-naming scheme. The precision of the analysis is effective in reducing the number of false positives issued by our tool.

A simple user interface. Users of our tool can find a variety of vulnerabilities involving tainted objects by specifying them using PQL [MLL05]. Our system provides a GUI auditing interface implemented on top of Eclipse, thus allowing users to perform security audits quickly during program development.

Experimental validation. We present a detailed experimental evaluation of our system and the static analysis approach on a set of large, widely-used open-source Java applications. We found a total of 29 security errors, including two important vulnerabilities in widely-used libraries. Eight out of nine of our benchmark applications had at least one vulnerability, and our analysis produced only 12 false positives.

1.5 Report Organization

The rest of this report is organized as follows. Section 2 presents a detailed overview of application-level security vulnerabilities we address. Section 3 describes our static analysis approach. Section 4 describes improvements that increase analysis precision and coverage. Section 5 describes the auditing environment our system provides. Section 6 summarizes our experimental findings. Section 7 describes related work, and Section 9 concludes. Finally, Appendix A summarizes information about Java API methods pertaining to vulnerabilities we find.

SECTION 2

Overview of Vulnerabilities

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 [Ope04b], 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 [Net04, Ope04b]. Recent reports include SQL injections in Oracle products [Lit03a] and cross-site scripting vulnerabilities in Mozilla Firefox [Kra05].

2.1 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. 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.

Further information on the relevant Java API methods is given in Appendix A and the benchmarks are described in Section 6.

2.2 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.

2.2.1 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.

2.2.2 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 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.

2.2.3 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.

2.2.4 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 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
```

```
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 2: 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`.

```
Content-type: application/
                x-www-form-urlencoded
Content-length: 100
```

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 [Ope04a]. □

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 [Ope04a].

2.2.5 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 4. Consider the HTTP `GET` request in Figure 3. 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].

2.2.6 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.

2.3 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.

2.3.1 SQL Injections

SQL injections first described in Section 2.1 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

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

Figure 3: An HTTP `GET` request containing a cookie.

of executable SQL. However, not using or improperly using prepared statements still leaves plenty of room for errors.

Example 5. Figure 2 shows two ways to update the salary of an employee, whose id is provided. The first method in Figure 2 (a) uses string concatenation to construct the query and leading to potential SQL injection attacks; the second in Figure 2 (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.

2.3.2 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 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. □

2.3.3 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"));
```

2.3.4 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 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. □

2.3.5 Command Injection

Command injection 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.

2.4 Secure Coding Practices

Clearly, all of the issues presented above are caused by unsafe coding techniques. Although user-provided data is typically validated on the client side, for example, using JavaScript validation routines for HTML form parameters before being being passed to the Web application, this sort of validation can be easily circumvented by an attacker by crafting either an HTTP request using one of widely available penetration testing tools [Chi04] or by inserting malicious parameter into the URL requested from the server. While client-side validation is still helpful to reject obviously invalid input, it is in no way a replacement of server-site checking.

Below we discuss some of the common prevention techniques commonly used by security-aware developers to avoid attacks based on insufficiently validated user input. In order to avoid attacks like SQL injections and cross-site scripting, all untrusted data must be properly validated before it is either passed to the database or output back to the browser. The following three approaches are widely-recognized strategies for protecting against malicious input [Ope04a]:

White-listing. (Accept Only Known Valid Data.) This is the preferred way to validate data. Applications should accept only input that is known to be safe and expected. As an example, lets assume a password reset system takes in usernames as input. Valid usernames would be de- fined as ASCII A-Z and 0-9. The application should check that the input is of type string, is comprised of A-Z and 0-9 (performing canonicalization checks as appropriate) and is of a valid length.

Black-listing. (Reject Known Bad Data.) The rejecting bad data strategy relies on the application knowing about specific malicious payloads. For instance, searching for JavaScript keywords passed in as part of input is one example of this strategy. While it is true that this strategy can limit exposure, it is very difficult for any application to maintain an up-to-date database of Web application attack signatures.

Sanitize All Input Data. Attempting to make bad data harmless is certainly an effective second line of defense, especially when dealing with rejecting bad input. However, the task of writing sanitization routines is a difficult one. Better widely available libraries are necessary so that developers do not have to develop their own sanitization routines. In fact, the errors we found in `blojsom` were due to sanitization routines that did not perform adequate checking.

SECTION 3

Static Analysis

In this section we present a static analysis that addresses the tainted object propagation problem described in Section 2.

3.1 Tainted Object Propagation

We start by defining the terminology that was informally introduced in Example 1. We define an *access path* as a sequence of field accesses, array index operations, or method calls separated by dots. For instance, the result of applying access path `f.g` to variable `v` is `v.f.g`. We denote the empty access path by ϵ; array indexing operations are indicated by [].

A *tainted object propagation problem* consists of a set of *source descriptors*, *sink descriptors*, and *derivation descriptors*: These descriptors formally specify how source methods in the program can generate unsafe 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 Java string manipulation routines.

- 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.

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.

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 input are often implemented using character-level string manipulation. Since taint does not propagate through such routines, they should not be included in the list of derivation descriptors.

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

Example 8. We can formulate the problem of detecting parameter tampering attacks that result in a SQL injection as follows: the source descriptor for obtaining parameters from an HTTP request is:

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

The 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:

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

We show only a few descriptors here; more information about the descriptors used in our experiments for different kinds of vulnerabilities can be found in Appendix A. □

Below we formally define a security violation:

Definition 3.1 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 3.2 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 3.3 Object o_2 is *derived* from object o_1, written $\mathit{derivedStream}(o_1, o_2)$, 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 3.4 An object is *tainted* if it is obtained by applying relation *derivedStream* to a source object zero or more times.

Definition 3.5 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:

$$\forall_{0 \leq i < k} \; i : \mathit{derivedStream}(o_i, o_{i+1}).$$

We refer to object pair $\langle o_1, o_k \rangle$ as a *source-sink pair*.

3.2 Specifications Completeness

The problem of obtaining a complete specification for a tainted object propagation problem is an important one. If a specification is incomplete, important errors will be missed even if we use a sound analysis that finds all vulnerabilities matching a specification. To come up with a list of source and sink descriptors for vulnerabilities in our experiments, we used the documentation of the relevant J2EE 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 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.

3.3 Static Analysis

Our approach is to use a sound static analysis to find all potential violations matching a vulnerability specification given by its source, sink, and derivation descriptors. To find security violations statically, it is necessary to know what *objects* these descriptors may refer to, a general problem known as *pointer* or *points-to analysis*.

3.3.1 Role of Pointer Information

To illustrate the need for points-to information, we consider the task of auditing a piece of Java code for SQL injections caused by parameter tampering, as described in Example 1.

Example 9. In the code below, string `param` is tainted because it is returned from a source method `getParameter`. So is `buf1`, because it is derived from `param` in the call to `append` on line 6. Finally, string `query` is passed to sink method `executeQuery`.

```
1   String param = req.getParameter("user");
2
3   StringBuffer buf1;
4   StringBuffer buf2;
5   ...
6   buf1.append(param);
7   String query = buf2.toString();
8   con.executeQuery(query);
```

Unless we know that variables `buf1` and `buf2` may *never* refer to the same object, we would have to conservatively assume that they may. Since `buf1` is tainted, variable `query` may also refer to a tainted object. Thus a conservative tool that lacks additional information about pointers will flag the call to `executeQuery` on line 8 as potentially unsafe. □

An unbounded number of objects may be allocated by the program at run time, so, to compute a finite answer, the pointer analysis statically approximates dynamic program objects with a finite set of static object "names". A common approximation approach is to name an object by its *allocation site*, which is the line of code that allocates the object.

3.3.2 Finding Violations Statically

Points-to information enables us to find security violations statically. Points-to analysis results are represented as the relation $\mathit{pointsto}(v, h)$, where v is a program variable and h is an allocation site in the program.

Definition 3.6 A *static security violation* is a sequence of heap allocation sites $h_1 \ldots h_k$ such that

1. There exists a variable v_1 such that $\mathit{pointsto}(v_1, h_1)$, where v_1 corresponds to access path p applied to argument n of a call to method m for a source descriptor $\langle m, n, p\rangle$.

2. There exists a variable v_k such that $\mathit{pointsto}(v_k, h_k)$, where v_k corresponds to applying access path p to argument n in a call to method m for a sink descriptor $\langle m, n, p\rangle$.

3. There exist variables $v_1, \ldots, v_k$ such that

$$\underset{1 \le i < k}{\forall} : \mathit{pointsto}(v_i, h_i) \wedge \mathit{pointsto}(v_{i+1}, h_{i+1}),$$

   where variable v_i corresponds to applying p_s to argument n_s and v_{i+1} corresponds applying p_d to argument n_d in a call to method m for a derivation descriptor $\langle m, n_s, p_s, n_d, p_d\rangle$.

Our static analysis is based on a context-sensitive Java points-to analysis developed by Whaley and Lam [WL04]. Their algorithm uses binary decision diagrams (BDDs) to efficiently represent and manipulate points-to results for exponentially many contexts in a program. They have developed a tool called `bddbddb` (BDD-Based Deductive DataBase) that automatically translates program analyses expressed in terms of Datalog [Ull89] (a language used in deductive databases) into highly efficient BDD-based implementations. The results of their points-to analysis can also be accessed easily using Datalog queries. Notice that in the absence of derived objects, finding security violations can be easily done with pointer analysis alone, because pointer analysis tracks objects as they are passed into or returned from methods.

However, it is relatively easy to implement the tainted object propagation analysis using `bddbddb`. Constraints of a specification as given by Definition 3.6 can be translated into Datalog queries straightforwardly. Facts such as "variable v is parameter n of a call to method m" map directly into Datalog relations representing the internal representation of the Java program. The points-to results used by the constraints are also readily available as Datalog relations after pointer analysis has been run.

The static analysis is fully interprocedural: calls to source, sink, and derivation methods may be located in different methods. It is important to point out that what violations are detected depends the portion of the call graph that is statically analyzed; however, determining classes that may be used at runtime is statically undecidable. Because Java supports dynamic loading and classes can be dynamically generated on the fly and called reflectively, we can find vulnerabilities only in the code available to the static analysis. For reflective calls, we use a simple analysis that handles common uses of reflection to increase the size of the analyzed call graph.

3.3.3 Role of Pointer Analysis Precision

Pointer analysis has been the subject of much compiler research over the last two decades. Because determining what heap objects a given program variable may point to during program execution is undecidable, sound analyses compute conservative approximations of the solution. Previous points-to approaches typically trade scalability for precision, ranging from highly scalable but imprecise techniques [Ste96] to precise approaches that have not been shown to scale [SRW99].

In the absence of precise information about pointers, a sound tool would conclude that many objects are tainted and hence report many false positives. Therefore, many practical tools use an unsound approach to pointers, assuming that pointers are unaliased unless proven otherwise [BPS00, HCXE02]. Such an approach, however, may miss important vulnerabilities.

Having precise points-to information can significantly reduce the number of false positives. Context sensitivity refers to the ability of an analysis to keep information from different invocation contexts of a method separate and is known to be an important feature contributing to precision. The effect of context sensitivity on analysis precision is illustrated by the example below.

```
1   class DataSource {
2       String url;
3       DataSource(String url) {
4           this.url = url;
5       }
6       String getUrl(){
7           return this.url;
8       }
9       ...
10  }
11  String passedUrl = request.getParameter("...");
12  DataSource ds1   = new DataSource(passedUrl);
13  String localUrl  = "http://localhost/";
14  DataSource ds2   = new DataSource(localUrl);
15
16  String s1        = ds1.getUrl();
17  String s2        = ds2.getUrl();
```

Figure 4: Example showing the importance of context sensitivity.

Example 10. Consider the code snippet in Figure 4. The class `DataSource` acts as a wrapper for a URL string. The code creates two `DataSource` objects and calls `getUrl` on both objects. A context-insensitive analysis would merge information for calls of `getUrl` on lines 16 and 17. The reference `this`, which is considered to be argument 0 of the call, points to the object on line 12 and 14, so `this.url` points to either the object returned on line 11 or `"http://localhost/"` on line 13. As a result, both `s1` and `s2` will be considered tainted if we rely on context-insensitive points-to results. With a context-sensitive analysis, however, only `s2` will be considered tainted. □

While many points-to analysis approaches exist, until recently, we did not have a scalable analysis that gives a conservative yet precise answer. The context-sensitive, inclusion-based points-to analysis by Whaley and Lam is both precise and scalable [WL04]. It achieves scalability by using BDDs to exploit the similarities across the exponentially many calling contexts.

A *call graph* is a static approximation of what methods may be invoked at all method calls in the program. While there are exponentially many acyclic call paths through the call graph of a program, the compression achieved by BDDs makes it possible to efficiently represent as many as 10^{14} contexts. The framework we propose in this paper is the first practical static analysis tool for security to leverage the BDD-based approach. The use of BDDs has allowed us to scale our framework to programs consisting of almost 1,000 classes.

3.4 Specifying Taint Problems in PQL

While a useful formalism, source, sink, and derivation descriptors as defined in Section 3.1 are not a user-friendly way to describe security vulnerabilities. Datalog queries, while giving the user complete control, expose too much of the program's internal representation to be practical. Instead, we use PQL, a program

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

Figure 5: Main query for finding source-sink pairs.

query language. PQL serves as syntactic sugar for Datalog queries, allowing users to express vulnerability patterns in a familiar Java-like syntax; translation of tainted object propagation queries from PQL into Datalog is straightforward. PQL is a general query language capable of expressing a variety of questions about program execution. However, we only use a limited form of PQL queries to formulate tainted object propagation problems.

We summarize only the most important features of PQL here; interested readers are referred to [MLL05] for a detailed description. In general, PQL can express many queries beyond tainted object propagation problems.

A PQL query is a pattern describing a sequence of dynamic events that involves variables referring to *dynamic object instances*. The **uses** clause declares all object variables the query refers to. The **matches** clause specifies the sequence of events on object variables that must occur for a match. Finally, the **return** clause specifies the objects returned by the query whenever a set of object instances participating in the events in the **matches** clause is found.

An important advantage of using PQL is that it *automatically* generates a pointer analysis-based Datalog query that can be used as a static checker for the properties of interest; these checkers are subsequently run to find potential vulnerabilities. PQL queries are first translated into queries in Datalog. Next, resolution of the resulting Datalog queries is performed using `bddbddb` [WL04], an efficient BDD-based solver which incorporates important optimizations that make query resolution fast.

PQL queries refer to dynamic objects, and points-to results provide a static approximation of what those objects might be. Pointer analysis is also per-

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

Figure 6: Transitive derived relation `derived*`.

```
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 7: PQL sub-queries for finding SQL injections.

formed within the bddbddb framework, and points-to results are used in PQL query translation as a link between dynamic objects and heap allocation sites in the program. However, using PQL allows us to largely hide the details of translation into Datalog and Datalog query resolution from the user.

3.4.1 Simple SQL Injection Query

Example 11. Figure 8 shows a PQL query for the SQL injection vulnerability in Example 1. This is a relatively simple query example that only addresses some SQL injections. The **uses** clause of a PQL query declares all objects used

```
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 8: The PQL query for finding simple SQL injections.

in the query. The **matches** 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 for string concatenation in Example 1. □

PQL queries are automatically translated into Datalog queries, which are in turn interpreted by `bddbddb`. As can be seen from the example below, the resulting Datalog is quite involved even for a relative simple query and is therefore not a very good specification language for describing vulnerabilities. The translation process is syntax-directed and is further described in [MLL05].

Example 12. The result of translating `simpleSQLInjection` into Datalog is shown below. Object x in PQL is approximated by allocated site h^x in Datalog. In addition to this, the following relations are used as part of translation:

- $\mathit{pointsto}(c, i, m)$ means that in context c, variable v points to heap allocation site h.
- $\mathit{actual}(i, v, n)$ means that variable v is the actual argument number n of call site i.

- $ret(i, v)$ means that variable v is returned at invocation site i.
- $call(c, i, m)$ means that m may be called at invocation site i in context c.

$$
\begin{aligned}
simpleSQLInjection(h^{param}, h^{derived})\ :-\ & \\
& ret(i_1, v_1), \\
& call(c_1, i_2, \texttt{"HttpServletRequest.getParameter"}), \\
& pointsto(c_1, v_1, h^{param}), \\
& \\
& actual(i_2, v_2, 0), actual(i_2, v_3, 1), \\
& call(c_2, i_2, \texttt{"StringBuffer.append"}), \\
& pointsto(c_2, v_2, h^{temp}), \\
& pointsto(c_2, v_3, h^{param}), \\
& \\
& actual(i_3, v_4, 0), ret(i_3, v_5), \\
& call(c_3, i_3, \texttt{"StringBuffer.toString"}), \\
& pointsto(c_3, v_4, h^{temp}), \\
& pointsto(c_3, v_5, h^{derived}), \\
& \\
& actual(i_4, v_6, 0), actual(i_4, v_7, 1), \\
& call(c_4, i_4, \texttt{"Connection.execute"}), \\
& pointsto(c_4, v_6, h^{con}), \\
& pointsto(c_4, v_7, h^{derived})\ .
\end{aligned}
$$

The same Datalog query modulo variable names may be obtained directly from the descriptors in Example 8. □

3.4.2 Queries for a Taint Problem

We illustrate the task of creating a taint problem by demonstrating what is involved in specifying SQL injection vulnerabilities caused by a variety of sources. Source-sink object pairs corresponding to static security violations for a given tainted object propagation problem are computed by query `main` in Figure 5. This query uses auxiliary queries `source` and `sink` used to define source and sink objects as well as query `derived`∗ shown in Figure 6 that captures a transitive derivation relation. Object `sourceObj` in `main` is returned by sub-query `source`. Object `sinkObj` is the result of sub-query `derived`⋆ with `sourceObj` used as a sub-query parameter and is also the result of sub-query `sink`. Therefore, `sinkObj` returned by query `main` matches all tainted objects that are also sink objects.

Semicolons are used in PQL to indicate a sequence of events that must occur in order. Sub-query `derived`∗ defines a transitive derived relation: object `y` is transitively derived from object `x` by applying sub-query `derived` zero or more times. This query takes advantage of PQL's sub-query mechanism to define a transitive closure recursively. Sub-queries `source`, `sink`, and `derived` are specific to a particular tainted object propagation problem, as shown in the example below.

Example 13. This example describes sub-queries `source`, `sink`, and `derived` shown in Figure 7 that can be used to match SQL injections, such as the one described in Example 1. Usually these sub-queries 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, sub-queries `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.

SECTION 4

Precision and Coverage Improvements

This section describes improvements we made to the object-naming scheme used in the original points-to analysis [WL04]. These improvements greatly increase the precision of the points-to results and reduce the number of false positives produced by our analysis and are further described in Section 4.1.1. In addition to difficulties involved in getting a low rate of false positives, which is a common issue with static analysis tools, Web applications present a unique set of challenges. In particular, it is not obvious what code needs to be analyzed. In Section 4.2.1 we present techniques designed to increase the coverage our our static technique.

4.1 Precision Improvements

The lack of precision is a common reason for why static analysis tools do not enjoy a wide adoption in practice. This is justified by the fact that a developer is rarely willing to examine tens or hundreds of false alarms to find a few “true” positives. In our work, we have focused on precision and identified two areas of our analysis where imprecise static treatment was responsible for a multitude of false positives. Below we describe our more precise handling of containers and string routines that allows us to achieve a significant increase in precision.

```
class Vector {
  Object[] table = new Object[1024];

  void add(Object value){
      int i    = ...;
      // optional resizing ...
      table[i] = value;
  }

   Object getFirst(){
       Object value = table[0];
       return value;
   }
 }
 String s1 = "...";
 Vector v1 =  new Vector();
 v1.add(s1);
 Vector v2 =  new Vector();
 String s2 = v2.getFirst();
```

Figure 9: Typical container definition and usage.

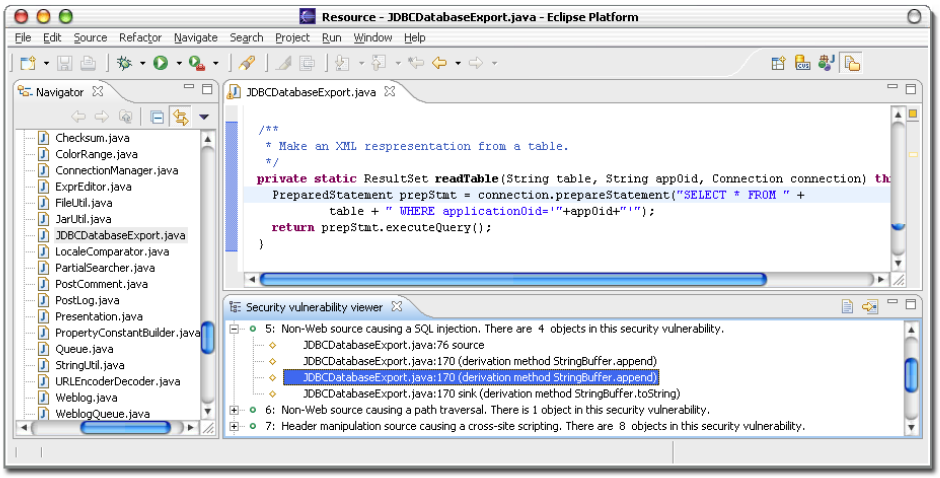

Figure 10: Tracking a SQL injection vulnerability in the Eclipse GUI plugin. Objects involved in the vulnerability trace are shown at the bottom.

4.1.1 Handling of Containers

Containers such as hash maps, vectors, lists, and others are a common source of imprecision in the original pointer analysis algorithm. The imprecision is due to the fact that objects are often stored in a data structure allocated *inside the container class definition*. As a result, the analysis cannot statically distinguish between objects stored in different containers.

Example 14. The abbreviated vector class in Figure 9 allocates an array called `table` on line 2 and vectors `v1` and `v2` share that array. As a result, the original analysis will conclude that the `String` object referred to by `s2` retrieved from vector `v2` may be the same as the `String` object `s1` deposited in vector `v1`. □

To alleviate this problem and improve the precision of the results, we create a new object name for the internally allocated data structure for every allocation site of the external container. This new name is associated with the allocation site of the underlying container object. As a result, the type of imprecision described above is eliminated and objects deposited in a container can only be retrieved from a container created at the same allocation site. In our implementation, we have applied this improved object naming to standard Java container classes including `HashMap`, `HashTable`, and `LinkedList`.

4.1.2 Handling of String Routines

Another set of methods that requires better object naming is Java string manipulation routines. Methods such as `String.toLowerCase`() allocate `String` objects that are subsequently returned. With the default object-naming scheme, all the allocated strings are considered tainted if such a method is ever invoked on a tainted string.

We alleviate this problem by giving unique names to results returned by string manipulation routines at different call sites. We currently apply this object naming improvement to Java standard libraries only. As explained in Section 6.4, imprecise object naming was responsible for *all* the 12 false positives produced by our analysis.

4.2 Coverage Improvements

In this section we describe changes to the static analysis that allow us to increase the amount of code that is analyzed statically.

4.2.1 Finding Root Methods in Web Applications

Our focus is Web applications, which are designed to be deployed within an application server. While analyzing the server *together* with the application is possible, in practice it is prohibitively expensive because of the size of a typical application server. Instead we chose to analyze the Web application in a standalone manner by providing a stub that emulates the environment in which the application is supposed to execute. This is similar to modeling the environment in model checking [TDP03] or using mock objects for testing [MFC00].

While finding all possible root methods in an application is generally a problem, Web applications present a somewhat unique challenge. J2EE-based applications are designed to run within a J2EE application server such as Apache Tomcat or IBM Websphere. A typical Web application we analyzed defines a set of *servlets* and *Struts actions* that are listed in a *deployment descriptor* parsed by the application server to determine what code to invoke. To include all the necessary servlets and actions in our analysis, we generate an *invocation stub*, a small driver program that invokes each servlet and action in the application in turn.

Methods of servlets and actions called from the invocation stub expect objects implementing interfaces `HttpServletRequest` and `HttpServletResponse` to be passed in as parameters. These interfaces are implemented by classes defined inside the application server that cannot be easily instantiated from a standalone program. In order to have concrete objects to pass to these methods in the invocation stub, we create our own "mock" versions of classes implementing these interfaces for the purpose of analysis [MFC00]. While this approach allows us to scale to large applications, we may miss some vulnerabilities contained in application server sources, which are not analyzed.

To generate an invocation stub, *web application descriptors* contained in file `web.xml` are parsed to find all servlets, filters, and listeners contained in the application. Similarly, calls are generated for *Struts actions*. An example of such a stub generated for `blueblog` is shown in Figure 11. Method `processServlets` constructs a mock `MyHttpServletRequest` and `MyHttpServletResponse` and passes them to method `service` of a newly constructed `BBServlet`.

```
package se.bluefish.blueblog;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpServlet;

import MyMockLib.MyHttpServletRequest;
import MyMockLib.MyHttpServletResponse;
import java.io.IOException;

class InvokeServlets {
    public static void main(String[] args) throws IOException {
        processServlets();
    }

    public static void processServlets() {
        try {
            HttpServletRequest  request  = new MyHttpServletRequest();
            HttpServletResponse response = new MyHttpServletResponse();

            se.bluefish.blueblog.servlet.BBServlet servlet =
                     new se.bluefish.blueblog.servlet.BBServlet();

            servlet.service(request, response);

        } catch (Exception e) {
            e.printStackTrace();
        }

        try {
            HttpServletRequest request   = new MyHttpServletRequest();
            HttpServletResponse response = new MyHttpServletResponse();

            se.bluefish.blueblog.servlet.ForwardingServlet servlet =
                     new se.bluefish.blueblog.servlet.ForwardingServlet();

            servlet.service(request, response);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Figure 11: Invocation stub program generated for `blueblog`.

4.2.2 Treatment of Reflection

The presence of reflection in Java complicates the analysis of Java programs considerably. Reflection is used to create new objects or call methods, given their names. The most common use of reflection is to dynamically create objects by name, following the coding idiom below:

```
String className = ...;
Class  c = Class.forName(className);
Object o = c.newInstance();
T      t = (T) o;
```

The call to `Class.forInstance` retrieves a class, whose name is specified by string `className`. The call to `newInstance` creates a new object of that class. However, not statically knowing what `className` is prevents the analysis from knowing what object may be instantiated at the call to `newInstance`.

While a full treatment of reflection is beyond this work, we augment the class construction process to find targets for `newInstance` calls. Statically determining what `className` may be is complicated by the presence of pointers. For each call to `Class.newInstance`, we

1. use pointer information to find all calls to `Class.forName` that may return `className`;

2. for each call to `Class.forName(s)`, find all constant class name strings that `s` may refer to;

3. for each constant string representing a class name `S` obtained in step 2, augment the call graph to include an edge from a call site of `c.newInstance` to `new S()`.

4.3 Soundness and Completeness

Our approach finds all vulnerabilities in the statically analyzed portion of the code. To find all potential security vulnerabilities, the user-supplied problem specification must be complete. Making sure that the sets of sources, derivation, and sink descriptors are complete is a difficult problem, however. Furthermore, all code that may be executed at runtime needs to be analyzed for errors: it is typical for Web-based Java applications to be shipped with a multitude of libraries (or jars); however, only a small percentage of classes are used during application execution. Therefore, analyzing all of the library classes is generally not practical. In order to compute the set of methods reachable at runtime, as described in Sections 4.2.1 and 4.2.2, *all* relevant root methods must be included and call sites must have *all* their targets resolved.

SECTION 5

Auditing Environment

The static analysis described in the previous two sections forms the basis of our security-auditing tool for Java applications. The tool allows a user to specify security patterns to detect. User-provided specifications are expressed as PQL queries, as described in Section 3.4. These queries are automatically translated into Datalog queries, which are subsequently resolved using `bddbddb`.

To help the user with the task of examining violation reports, our provides an intuitive GUI interface. The interface is built on top of Eclipse, a popular open-source Java development environment. As a result, a Java programmer can assess the security of his application, often without leaving the development environment used to create the application in the first place.

A typical auditing session involves applying the analysis to the application and then exporting the results into Eclipse for review. Our Eclipse plugin allows the user to easily examine each vulnerability by navigating among the objects involved in it. Clicking on each object allows the user to navigate through the code displayed in the text editor in the top portion of the screen.

Example 15. An example of using the Eclipse GUI is shown in Figure 10. The bottom portion of the screen lists all potential security vulnerabilities reported by our analysis. One of them, a SQL injection caused by non-Web input is expanded to show all the objects involved in the vulnerability. The source object on line 76 of `JDBCDatabaseExport.java` is passed to derived objects using derivation methods `StringBuffer.append` and `StringBuffer.toString` until it reaches the sink object constructed and used on line 170 of the same file. Line 170, which contains a call to `Connection.prepareStatement`, is highlighted in the Java text editor shown on top of the screen. □

SECTION 6

Experimental Results

In this section we summarize the experiments we performed and described the security violations we found. We start out by describing our benchmark applications and experimental setup, describe some representative vulnerabilities found by our analysis, and analyze the impact of analysis features on precision.

6.1 Benchmark Applications

While there is a fair number of commercial and open-source tools available for testing Web application security, there are no established benchmarks for comparing tools' effectiveness. The task of finding suitable benchmarks for our experiments was especially complicated by the fact that most Web-based applications are proprietary software, whose vendors are understandably reluctant to reveal their code, not to mention the vulnerabilities found. At the same time, we did not want to focus on artificial micro-benchmarks or student projects that lack the complexities inherent in real applications. While some attempts have been made at constructing artificial benchmarks [Fou, Pro], we believe that real-life programs are much better suited for testing security tools. We focused on a set of large, representative open-source Web-based J2EE applications, most of which are available on SourceForge.

In the course of our research in application security at Stanford, our group has developed a suite of benchmarks called SecuriBench [Liv05]. Thus far it consists of 8 real-life open-source Web-based applications written in Java and developed on top of J2EE. Most programs are medium-sized, with the larger ones consisting of almost 200,000 lines of code. We are making these benchmarks publicly available in hopes of fostering collaboration between researchers. These benchmarks can serve as test cases for researchers and industry practitioners working in this area.

The benchmark applications we used are briefly described below. `jboard`, `blueblog`, `blojsom`, `personalblog`, `snipsnap`, `pebble`, and `roller` are Web-based bulletin board and blogging applications. `webgoat` is a J2EE application designed by the Open Web Application Security Project [Ope04a, Ope04b] as a test case and a teaching tool for Web application security. Finally, `road2hibernate` is a test program developed for `hibernate`, a popular object persistence library, which is not a Web application and is not therefore part of SecuriBench.

Applications were selected from among J2EE-based open-source projects on SourceForge solely on the basis of their size and popularity. Other than `webgoat`, which we knew had intentional security flaws, we had no prior knowledge as to whether the applications had security vulnerabilities. Most of our benchmark applications are used widely: `roller` is used on dozens of sites including prominent ones such as `blogs.sun.com`. `snipsnap` has more than 50,000 downloads

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

Figure 12: Summary of information about the benchmarks. Applications are sorted by the total number of analyzed classes.

according to its authors. `road2hibernate` is a wrapper around `hibernate`, a highly popular object persistence library that is used by multiple large projects, including a news aggregator and a portal. `personalblog` has more than 3,000 downloads according to SourceForge statistics. Finally, `blojsom` was adopted as a blogging solution for the Apple Tiger Weblog Server.

Figure 12 summarizes information about our benchmark applications. Notice that the traditional lines-of-code metric is somewhat misleading in the case of applications that use large libraries. Many of these benchmarks depend on massive libraries, so, while the application code may be small, the full size of the application executed at runtime is quite large. An extreme case is `road2hibernate`, which is a small 140-line stub program designed to exercise the `hibernate` object persistence library; however, the total number of analyzed classes for `road2hibernate` exceeded 800. A better measure is given in the last column of Figure 12, which shows the total number of classes in each application's call graph.

6.2 Experimental Setup

The implementation of our system is based on the `joeq` Java compiler and analysis framework. In our system we use a translator from PQL to Datalog [MLL05] and the `bddbddb` program analysis tool [WL04] to find security violations. We applied static analysis to look for all tainted object propagation problems described in this report, and we used a total of 28 source, 18 sink, and 29 derivation descriptors in our experiments. The derivation descriptors correspond to methods in classes such as `String`, `StringBuffer`, `StringTokenizer`, etc. Source and sink descriptors correspond to methods declared in 19 different J2EE classes, as is further described in Appendix A.

We used four different variations of our static analysis, obtained by either enabling or disabling context sensitivity and improved object naming. Analysis times for the variations are listed in Figure 13. Running times shown in the table

	Pre-processing	Points-to analysis				Taint analysis			
Context sensitivity				✓	✓			✓	✓
Improved naming			✓		✓		✓		✓
`jboard`	37	8	7	12	10	14	12	16	14
`blueblog`	39	13	8	15	10	17	14	21	16
`webgoat`	57	45	30	118	90	69	66	106	101
`blojsom`	60	18	13	25	16	24	21	30	27
`personalblog`	173	107	28	303	32	62	50	19	59
`snipsnap`	193	58	33	142	47	194	154	160	105
`road2hibernate`	247	186	40	268	43	73	44	161	58
`pebble`	177	58	35	117	49	150	140	136	100
`roller`	362	226	55	733	103	196	83	338	129

Figure 13: Summary of times, in seconds, it takes to perform preprocessing, points-to, and taint analysis for each analysis variation. Analysis variations have either context sensitivity or improved object naming enabled, as indicated by ✓ signs in the header row.

are obtained on an Opteron 150 machine with 4 GB of memory running Linux. The first section of the table shows the times to pre-process the application to create relations accepted by the pointer analysis; the second shows points-to analysis times; the last presents times for the tainted object propagation analysis.

It should be noted that the taint analysis times often *decrease* as the analysis precision increases. Contrary to intuition, we actually pay *less* for a more precise analysis. Imprecise answers are big and therefore take a long time to compute and represent. In fact, the context-insensitive analysis with default object naming runs significantly slower on the largest benchmarks than the most precise analysis. The most precise analysis version takes a total of less than 10 minutes on the largest application; we believe that this is acceptable given the quality of the results the analysis produces.

6.3 Vulnerabilities Discovered

The static analysis described in this report reports a total of 41 potential security violations in our nine benchmarks, out of which 29 turn out to be security errors, while 12 are false positives. All but one of the benchmarks had at least one security vulnerability. Moreover, except for errors in `webgoat` and one HTTP splitting vulnerability in `snipsnap` [Gen04], none of these security errors had been reported before.

6.3.1 Validating the Errors We Found

Not all security errors found by static analysis or code reviews are necessarily *exploitable* in practice. The error may not correspond to a path that can be taken dynamically, or it may not be possible to construct meaningful malicious input. Exploits may also be ruled out because of the particular configuration of the application, but configurations may change over time, potentially making exploits possible. For example, a SQL injection that may not work on one database may become exploitable when the application is deployed with a database system that does not perform sufficient input checking. Furthermore, virtually all static errors we found can be fixed easily by modifying several lines of Java source code, so there is generally no reason *not* to fix them in practice.

After we ran our analysis, we manually examined all the errors reported to make sure they represent security errors. Since our knowledge of the applications was not sufficient to ascertain that the errors we found were exploitable, to gain additional assurance, we reported the errors to program maintainers. We only reported to application maintainers only those errors found in the *application code* rather than general libraries over which the maintainer had no control. Almost all errors we reported to program maintainers were confirmed, resulting in more that a dozen code fixes.

Because `webgoat` is an artificial application designed to contain bugs, we did not report the errors we found in it. Instead, we dynamically confirmed some of the statically detected errors by running `webgoat`, as well as a few other benchmarks, on a local server and creating actual exploits.

It is important to point out that our current analysis ignores control flow. Without analyzing the predicates, our analysis may not realize that a program has checked its input, so some of the reported vulnerabilities may turn out to be false positives. However, our analysis shows all the steps involved in propagating taint from a source to a sink, thus allowing the user to check if the vulnerabilities found are exploitable.

Many Web-based application perform some form of input checking. However, as in the case of the vulnerabilities we found in `snipsnap`, it is common that some checks are missed. It is surprising that our analysis did not generate any false warnings due to the lack of predicate analysis, even though many of the applications we analyze include checks on user input. Two security errors in `blojsom` flagged by our analysis deserve special mention. The user-provided input *was* in fact checked, but the validation checks were too lax, leaving room for exploits. Since the sanitization routine in `blojsom` was implemented using string operations as opposed to direct character manipulation, our analysis detected the flow of taint from the routine's input to its output. To prove the vulnerability to the application maintainer, we created an exploit that circumvented all the checks in the validation routine, thus making path-traversal vulnerabilities possible. Note that if the sanitation was properly implemented, our analysis would have generated some false positives in this case.

6.3.2 Classification of Errors

This section presents a classification of all the errors we found. A summary of our experimental results is presented in Figure 14(a). Columns 2 and 3 list the number of source and sink objects for each benchmark. It should be noted that the number of sources and sinks for all of these applications is quite large, which suggests that security auditing these applications is time-consuming, because the time a manual security code review takes is roughly proportional to the number of sources and sinks that need to be considered. The table also shows the number of vulnerability reports, the number of false positives, and the number of errors for each analysis version.

Figure 15 presents a classification of the 29 security vulnerabilities we found grouped by the type of the source in the table rows and the sink in table columns. For example, the cell in row 4, column 1 indicates that there were 2 potential SQL injection attacks caused by non-Web sources, one in `snipsnap` and another in `road2hibernate`.

Overall, parameter manipulation was the most common technique to inject malicious data (13 cases) and HTTP splitting was the most popular exploitation technique (11 cases). Many HTTP splitting vulnerabilities are due to an unsafe programming idiom where the application redirects the user's browser to a page whose URL is user-provided as the following example from `snipsnap` demonstrates:

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

Most of the vulnerabilities we discovered are in application code as opposed to libraries. While errors in application code may result from simple coding mistakes made by programmers unaware of security issues, one would expect library code to generally be better tested and more secure. Errors in libraries expose all applications using the library to attack. Despite this fact, we have managed to find two attack vectors in libraries: one in a commonly used Java library `hibernate` and another in the J2EE implementation. While a total of 29 security errors were found, because the same vulnerability vector in J2EE is present in four different benchmarks, they actually corresponded to 26 *unique* vulnerabilities.

6.3.3 SQL Injection Vector in `hibernate`

We start by describing a vulnerability vector found in `hibernate`, an open-source object-persistence library commonly used in Java applications as a lightweight back-end database. `hibernate` provides the functionality of saving program data structures to disk and loading them at a later time. It also allows applications to search through the data stored in a `hibernate` database. Three programs in our benchmark suite, `personalblog`, `road2hibernate`, and `snipsnap`, use `hibernate` to store user data.

We have discovered an attack vector in code pertaining to the search functionality in `hibernate`, version 2.1.4. The implementation of method `Session.find` retrieves objects from a `hibernate` database by passing its input string argument through a sequence of calls to a SQL execute statement. As a result, all invocations of `Session.find` with unsafe data, such as the two errors we found in `personalblog`, may suffer from SQL injections. A few other public methods such as `iterate` and `delete` also turn out to be attack vectors.

This situation illustrates a more general pattern: an attack vector in a commonly used software component can lead to vulnerabilities in all of the clients of that component. Our findings highlight the importance of securing commonly used software components in order to protect their clients.

6.3.4 Cross-site Tracing Attacks

Analysis of `webgoat` and several other applications revealed a previously unknown vulnerability in core J2EE libraries, which are used by thousands of Java applications. This vulnerability pertains to the `TRACE` method specified in the HTTP protocol. `TRACE` is used to echo the contents of an HTTP request back to the client for debugging purposes. However, the contents of user-provided headers are sent back verbatim, thus enabling cross-site scripting attacks.

In fact, this variation of cross-site scripting caused by a vulnerability in HTTP protocol specification was discovered before, although the fact that it was present in J2EE was not previously announced. This type of attack has been dubbed *cross-site tracing* and it is responsible for CERT vulnerabilities 244729, 711843, and 728563. Because this behavior is specified by the HTTP protocol, there is no easy way to fix this problem at the source level. General recommendations for avoiding cross-site tracing include disabling `TRACE` functionality on the server or disabling client-side scripting [Gro03].

6.4 Analysis Features and False Positives

The version of our analysis that employs both context sensitivity and improved object naming described in Section 4 achieves very precise results, as measured by the number of false positives. In this section we examine the contribution of each feature of our static analysis approach to the precision of our results. We also explain the causes of the remaining 12 false positives reported by the most precise analysis version. To analyze the importance of each analysis feature, we examined the number of false positives as well as the number of tainted objects reported by each variation of the analysis. Just like false positives, tainted objects provide a useful metric for analysis precision: as the analysis becomes more precise, the number of objects deemed to be tainted decreases.

Figure 14(a) summarizes the results for the four different analysis versions. The first part of the table shows the number of tainted objects reported by the analysis. The second part of the table shows the number of reported security violations. The third part of the table summarizes the number of false positives. Finally, the last column provides the number of real errors detected for each benchmark. Figure 14(b) provides a graphical representation of the number of tainted objects for different analysis variations. Below we summarize our observations.

Context sensitivity combined with improved object naming achieves a very low number of false positives. In fact, the number of false positives was 0 for all applications but `snipsnap`. For `snipsnap`, the number of false positives was reduced more than 50-fold compared to the context-insensitive analysis version with no naming improvements. Similarly, not counting the small program `jboard`, the most precise version on average reported 5 times fewer tainted objects than the least precise. Moreover, the number of tainted objects dropped more that 15-fold in the case of `roller`, our largest benchmark.

To achieve a low false-positive rate, *both* context sensitivity and improved object naming are necessary. The number of false positives remains high for most programs when *only* one of these analysis features is used. One way to interpret the importance of context sensitivity is that the right selection of object "names" in pointer analysis allows context sensitivity to produce precise results. While it is widely recognized in the compiler community that special treatment of containers is necessary for precision, improved object naming *alone* is not generally sufficient to completely eliminate the false positives.

All 12 of the false positives reported by the most precise version for our analysis were located in `snipsnap` and were caused by insufficient precision of the default allocation site-based object-naming scheme. The default naming caused an allocation site in `snipsnap` to be conservatively considered tainted because a tainted object could propagate to that allocation site. The allocation site in question is located within `StringWriter.toString()`, a JDK function similar to `String.toLowerCase()` that returns a tainted `String` only if the underlying `StringWriter` is constructed from a tainted string. Our analysis conservatively concluded that the return result of this method may be tainted, causing a vulnerability to be reported, where none can occur at runtime. We should mention that *all* the false positives in `snipsnap` are eliminated by creating a new object name at every call to, `StringWriter.toString()`, which is achieved with a *one-line change* to the pointer analysis specification.

	Sources	Sinks	Tainted objects				Reported warnings				False positives				Errors
Context sensitivity					✓	✓			✓	✓			✓	✓	
Improved object naming				✓		✓		✓		✓		✓		✓	
`jboard`	1	6	268	23	2	2	0	0	0	0	0	0	0	0	0
`blueblog`	6	12	17	17	17	17	1	1	1	1	0	0	0	0	1
`webgoat`	13	59	1,166	201	903	157	51	7	51	6	45	1	45	0	6
`blojsom`	27	18	368	203	197	112	48	4	26	2	46	2	24	0	2
`personalblog`	25	31	2,066	1,023	1,685	426	460	275	370	2	458	273	368	0	2
`snipsnap`	155	100	1,168	791	897	456	732	93	513	27	717	78	498	12	15
`road2hibernate`	1	33	2,150	843	1,641	385	18	12	16	1	17	11	15	0	1
`pebble`	132	70	1,403	621	957	255	427	211	193	1	426	210	192	0	1
`roller`	32	64	2,367	504	1,923	151	378	12	261	1	377	11	260	0	1
Total	392	393	10,973	4,226	8,222	1,961	2,115	615	1,431	41	2,086	586	1,402	12	29

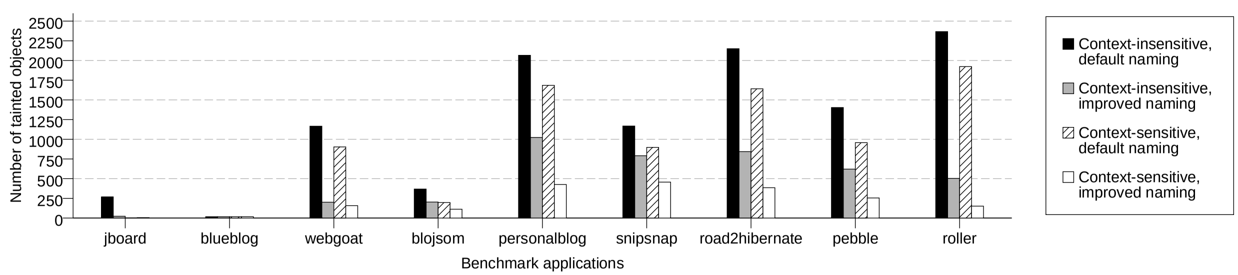

Figure 14: (a) Summary of data on the number of tainted objects, reported security violations, and false positives for each analysis version. Enabled analysis features are indicated by ✓ signs in the header row. **(b)** Comparison of the number of tainted objects for each version of the analysis.

	SQL injections	HTTP splitting	Cross-site scripting	Path traversal	Total
Header manip.	0	snipsnap = 6	blueblog: 1, webgoat: 1, pebble: 1, roller: 1 = 4	0	10
Parameter manip.	webgoat: 4, personalblog: 2 = 6	snipsnap = 5	0	blojsom = 2	13
Cookie poisoning	webgoat = 1	0	0	0	1
Non-Web inputs	snipsnap: 1, road2hibernate: 1 = 2	0	0	snipsnap = 3	5
Total	9	11	4	5	29

Figure 15: Classification of vulnerabilities we found. Each cell corresponds to a combination of a source type (in rows) and sink type (in columns).

SECTION 7

Related Work

While much attention has been given to the topic of detecting security errors in Web-based applications, applying powerful static analysis techniques to the problem is new. In addition to manual code reviews, which are commonly employed for finding vulnerabilities, the two most commonly used approaches are *penetration testing* and *runtime monitoring*, described below in Sections 7.1 and 7.2. We also review the relevant literature on static analysis for security in Section 7.3.

7.1 Penetration Testing

Current practical solutions for detecting Web application security problems generally fall into the realm of penetration testing [ASM05, BLT02, GH02, MJ03, SS02]. Penetration testing involves attempting to exploit vulnerabilities in a Web application or crashing it by coming up with a set of appropriate malicious input values. Penetration reports usually include a list of identified vulnerabilities [Imp]. However, this approach is incomplete. A penetration test can usually reveal only a small sample of all possible security risks in a system without identifying the parts of the system that have not been adequately tested. Generally, there are no standards that define which tests to run and which inputs to try. In most cases this approach is not effective and considerable program knowledge is needed to find application-level security errors successfully.

In order to increase coverage, some recent work has tried to automate finding test cases for Web applications [OWDH04]. To simplify the job of a penetration tester, multiple *fuzzing* tools are available. Fuzzing is a testing technique that generates and submits random or sequential data to various areas of an application in order to uncover vulnerabilities [HM04, MFS90].

7.2 Runtime Monitoring

A variety of both free and commercial runtime monitoring tools for evaluating Web application security are available. Proxies intercept HTTP and HTTPS data between the server and the client, so that data, including cookies and form fields, can be examined and modified, and resubmitted to the application [Chi04, Ope04c]. Commercial application-level firewalls available from NetContinuum, Imperva, Watchfire, and other companies take this concept further by creating a model of valid interactions between the user and the application and warning about violations of this model. Some application-level firewalls are based on signatures that guard against known types of attacks. The white-listing approach specifies what the valid inputs are; however, maintaining the rules for white-listing is challenging. In contrast, our technique can prevent security errors *before* they have a chance to manifest themselves.

7.3 Static Analysis Approaches

A good overview of static analysis approaches applied to security problems is provided in [CM04]. Simple lexical approaches employed by scanning tools such as ITS4 and RATS use a set of predefined patterns to identify potentially dangerous areas of a program [WK02]. While a significant improvement on Unix `grep`, these tools, however, have no knowledge of how data propagates throughout the program and cannot be used to automatically and fully solve taint-style problems.

A few projects use path-sensitive analysis to find errors in C and C++ programs [BPS00, HCXE02, LL03]. While capable of addressing taint-style problems, these tools rely on an unsound approach to pointers and may therefore miss some errors. The WebSSARI project uses combined unsound static and dynamic analysis in the context of analyzing PHP programs [HYH+04]. WebSSARI has successfully been applied to find many SQL injection and cross-site scripting vulnerabilities in PHP code.

An analysis approach that uses type qualifiers has been proven successful in finding security errors in C for the problems of detecting format string violations and user/kernel bugs [JW04, STFW01]. Context sensitivity significantly reduces the rate of false positives encountered with this technique; however, it is unclear how scalable the context-sensitive approach is.

Much of the work in information-flow analysis uses a type-checking approach, as exemplified by JFlow [Mye99]. Source annotations are required, and security is enforced by type checking. The compiler reads a program containing labeled types and, in checking the types, ensures that the program cannot contain improper information flow at runtime. The security type system in such a language enforces information-flow policies. The annotation effort, however, may be prohibitively expensive in practice. In addition to explicit information flows our approach addresses, JFlow also deals with implicit information flows.

Static analysis has been applied to analyzing SQL statements constructed in Java programs that may lead to SQL injection vulnerabilities [GSD04, WS04]. That work analyzes strings that represent SQL statements to check for potential type violations and tautologies. This approach assumes that a *flow graph* representing how string values can propagate through the program has been constructed a priori from points-to analysis results. However, since accurate pointer information is necessary to construct an accurate flow graph, it is unclear whether this technique can achieve the scalability and precision needed to detect errors in large systems.

SECTION 8

Future Work

The main claim of our approach is that we are able to find *all* vulnerabilities captured by a user-provided specification in the statically analyzed portion of the code. While this formulation provides guarantees that go beyond an unsound approach, it also leaves room for improvement, which justifies much of our future work. We are improving our framework in the following areas:

- **Improving static coverage.** As mentioned in Section 4.2, it is important to analyze all code that may be relevant at runtime statically. We are working on a comprehensive solution to construct a call graph in the presence of reflection. Methods such as `Class.forName` and `Class.newInstance` used for object creation are not the only ones that need to be analyzed. Java reflection APIs allow method invocation and object field manipulation through reflective APIs as well.

  Certain issues in call graph construction are specific to Web applications only, though. For instance, classes that are generated by the application server at runtime from Java Server Pages (JSPs). To obtain full coverage, these classes need to be pre-compiled and available for analysis.

- **Analysis of character-level manipulation.** As mentioned in Section 3.1, our framework stops tracking tainted data at the level of characters. While this is considerably less common that in C, character-level manipulation is still used in Java applications and some flows of tainted data are lost because of omitting it. We are developing analyses that would address common cases of character manipulation.

- **Improving specification completeness.** Our approach places the burden of coming up with a specification for the vulnerabilities of interest on the end-user.

  While our basic approach is sound, when applied to a particular set of taint problems, our analysis only find *all application vulnerabilities* as long as the problem specification is complete. Namely, for a particular taint problem, care must be taken to ensure that the sets of source, sink, and derivation descriptors are complete. If a particular source is missing, potential vulnerabilities caused by this source, if any, may be missed. If a particular descriptor is omitted from the user-provided specification, propagation of taint may be stopped prematurely thus potentially also missing some vulnerabilities. While the problem of inferring specifications in the general case is future work, we have applied some simple strategies to find source, sink, and derivation descriptors.

  We have created an analysis of what the sources are by following the flow of information *forward* from data retrieve from primitive socket routines. This allows us to find some sources that were previously missing in user-provided specification. In the process of solving the taint problem, we compute a set of "stuck" objects—objects that have no objects derived from them. Next we examine all methods into which stuck objects are passed. Some of these methods turn out to be obscure derivation methods missing in our initial specification, which we subsequently added. However, other methods we discovered were *sanitization methods*, whose purpose is to remove all characters leading to malicious exploits from the user-provided input.

SECTION 9

Conclusions

We showed how a general class of security errors in Java applications can be formulated as instances of the general *tainted object propagation* problem, which involves finding all *sink objects* derivable from *source objects* via a set of given *derivation rules*. We developed a precise and scalable analysis for this problem based on a precise context-sensitive pointer alias analysis and introduced extensions to the handling of strings and containers to further improve the precision. Our approach finds all vulnerabilities matching the specification within the statically analyzed code. Note, however, that errors may be missed if the user-provided specification is incomplete.

We formulated a variety of widespread vulnerabilities including SQL injections, cross-site scripting, HTTP splitting attacks, and other types of vulnerabilities as tainted object propagation problems. Our experimental results showed that our analysis is an effective practical tool for finding security vulnerabilities. We were able to find a total of 29 security errors, and all but one of our nine large real-life benchmark applications were vulnerable. Two vulnerabilities were located in commonly used libraries, thus subjecting applications using the libraries to potential vulnerabilities. Most of the security errors we reported were confirmed as exploitable vulnerabilities by their maintainers, resulting in more than a dozen code fixes. The analysis reported false positives for only one application. We determined that the false warnings reported can be eliminated with improved object naming.

SECTION 10

Acknowledgements

We are grateful to Michael Martin for his help with PQL and dynamic validation of some of the vulnerabilities we found and to John Whaley for his support with the `bddbddb` tool and the `joeq` framework. We thank our Usenix paper shepherd R. Sekar, whose insightful comments helped improve this paper considerably.

We thank the benchmark application maintainers for responding to our bug reports. We thank Amit Klein for providing detailed clarifications about Web application vulnerabilities and Ramesh Chandra, Chris Unkel, and Ted Kremenek and the anonymous paper reviewers for providing additional helpful comments. Finally, this material is based upon work supported by the National Science Foundation under Grant No. 0326227.

References

[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.

[ASM05] Brad Arkin, Scott Stender, and Gary McGraw. Software penetration testing. *IEEE Security and Privacy*, 3(1):84–87, 2005.

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

[Bea03] Kevin Beaver. Achieving Sarbanes-Oxley compliance for Web applications through security testing. http://www.spidynamics.com/support/whitepapers/WI_SOXwhitepaper.pdf, 2003.

[BLT02] Brian Buege, Randy Layman, and Art Taylor. *Hacking Exposed: J2EE and Java: Developing Secure Applications with Java Technology*. McGraw-Hill/Osborne, 2002.

[BPS00] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software - Practice and Experience (SPE)*, 30:775–802, 2000.

[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.

[CM04] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6):76–79, 2004.

[Com02] Computer Security Institute. Computer crime and security survey. http://www.gocsi.com/press/20020407.jhtml?_requestid=195148, 2002.

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

[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 7th USENIX Security Conference*, pages 63–78, January 1998.

[DFK+04] Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *Java Developer's Guide to Eclipse*. Addison-Wesley Professional, 2004.

[Fou] Foundstone, Inc. Hackme books, test application for Web security. http://www.foundstone.com/index.htm?subnav=resources/navigation.htm&subcontent=/resources/proddesc/hacmebooks.htm.

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

[Gen04] Gentoo Linux Security Advisory. SnipSnap: HTTP response splitting. http://www.gentoo.org/security/en/glsa/glsa-200409-23.xml, 2004.

[GH02] Daniel Geer and John Harthorne. Penetration testing: A duet. http://www.acsac.org/2002/papers/geer.pdf, 2002.

[Gro03] Jeremiah Grossman. Cross-site tracing (XST): The new techniques and emerging threats to bypass current Web security measures using TRACE and XSS. http://www.cgisecurity.com/whitehat-mirror/WhitePaper_screen.pdf, 2003.

[Gro04] Jeremiah Grossman. WASC activities and U.S. Web application security trends. http://www.whitehatsec.com/presentations/WASC_WASF_1.02.pdf, 2004.

[GSD04] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 645–654, 2004.

[HCXE02] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language Design and Implementation*, pages 69–82, 2002.

[HL01] Michael Howard and David LeBlanc. *Writing Secure Code*. Microsoft Press, 2001.

[HM04] Greg Hoglund and Gary McGraw. *Exploiting Software : How to Break Code*. Addison-Wesley Publishing, 2004.

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

[Hul01] George Hulme. New software may improve application security. http://www.informationweek.com/story/IWK20010209S0003, 2001.

[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 13th conference on World Wide Web*, pages 40–52, 2004.

[Imp] Imperva, Inc. SuperVeda penetration test. http://www.imperva.com/download.asp?id=3.

[JW04] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 2004 Usenix Security Conference*, pages 119–134, 2004.

[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.

[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.

[Liv05] Benjamin Livshits. Stanford SecuriBench. http://suif.stanford.edu/~livshits/securibench/, 2005.

[LL03] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 317–326, September 2003.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the 14th Usenix Security Symposium*, pages 271 – 286, August 2005.

[MFC00] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: Unit testing with mock objects. In *Proceedings of eXtreme Programming and Flexible Processes in Software Engineering*, May 2000.

[MFS90] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.

[MJ03] Jody Melbourne and David Jorm. Penetration testing for Web applications. http://www.securityfocus.com/infocus/1704, 2003.

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

[MRM03] James S. Miller, Susann Ragsdale, and Jim Miller. *The Common Language Infrastructure Annotated Standard*. Addison-Wesley Professional, 2003.

[Mye99] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, January 1999.

[Net04] NetContinuum, Inc. The 21 primary classes of Web application threats. https://www.netcontinuum.com/securityCentral/TopThreatTypes/index.cfm, 2004.

[Ope04a] Open Web Application Security Project. A guide to building secure Web applications. http://voxel.dl.sourceforge.net/sourceforge/owasp/OWASPGuideV1.1.pdf, 2004.

[Ope04b] Open Web Application Security Project. The ten most critical Web application security vulnerabilities. http://umn.dl.sourceforge.net/sourceforge/owasp/OWASPTopTen2004.pdf, 2004.

[Ope04c] Open Web Application Security Project. WebScarab. http://www.owasp.org/software/webscarab.html, 2004.

[OWDH04] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong Huang. Bypass testing of Web applications. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering*, 2004.

[Pro] The Open Web Application Security Project. WebGoat Project. http://www.owasp.org/software/webgoat.html.

[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.

[SRW99] S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 105–118, January 1999.

[SS02] Joel Scambray and Mike Shema. *Web Applications (Hacking Exposed)*. Addison-Wesley Professional, 2002.

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

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23th ACM Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 2001 Usenix Security Conference*, pages 201–220, August 2001.

[TDP03] Oksana Tkachuk, Matthew B. Dwyer, and Corina S. Pasareanu. Automated environment generation for software model checking. In *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, page 116, 2003.

[Ull89] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, Md., volume II edition, 1989.

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

[Web04] WebCohort, Inc. Only 10% of Web applications are secured against common hacking techniques. http://www.imperva.com/company/news/2004-feb-02.html, 2004.

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

[WK02] John Wilander and Mariam Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of 7th Nordic Workshop on Secure IT Systems*, November 2002.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, pages 131–144, June 2004.

[WS04] Gary Wassermann and Zhendong Su. An analysis framework for security in Web applications. In *Proceedings of the Specification and Verification of Component-Based Systems Workshop*, October 2004.

SECTION A

Source, Sink, and Derivation Descriptors

In this Appendix we provide information about source, sink, and derivation descriptors used in this paper. Source descriptors are presented in Figure 16, sink descriptors in Figure 17, and derivation descriptors in Figure 18.

Vulnerability type	Source descriptors = $\langle m, n, p\rangle$		
	Method m	**Parameter** n	**Access path** p
Parameter tampering attacks, hidden field manipulation, URL tampering	getParameter	-1	ϵ
	getParameters	-1	[]
	getParameterMap	-1	keySet().iterator().next()
	getParameterMap	-1	values().iterator().next()
	getParameterValues	-1	[]
	getQueryString	-1	ϵ
	getRemoteUser	-1	ϵ
	getComment	-1	ϵ
	getRequestedSessionId	-1	ϵ
	getProtocol	-1	ϵ
	getDomain	-1	ϵ
	getName	-1	ϵ
	getValue	-1	ϵ
HTTP header manipulation	getHeader	-1	ϵ
	getHeaders	-1	nextElement()
Cookie poisoning attacks	getCookies	-1	[].getPath(), [].getDomain(),
	getCookies	-1	[].getComment(), [].getValue(),
	getCookies	-1	[].getName()
Non-Web sources	*.main(String[])	1	[]

Figure 16: Summary of source descriptors for different types of vulnerabilities. All methods above are declared in class javax.servlet.ServletRequest.

Vulnerability type	Sink descriptors = $\langle m, n, p \rangle$ Method m	Parameter n	Access path p
SQL Injection	In package `java.sql`:		
	`Statement.executeUpdate(String, ...)`	1	ϵ
	`Statement.executeQuery(String)`	1	ϵ
	`Statement.execute(String, ...)`	1	ϵ
	`Statement.addBatch(String)`	1	ϵ
	`Connection.prepareStatement(String, ...)`	1	ϵ
	`Connection.prepareCall(String, ...)`	1	ϵ
Cross-site scripting, HTTP response splitting	In package `javax.servlet`:		
	`http.HttpServletResponse.sendError(int, String)`	1	ϵ
	`ServletOutputStream.print(String)`	1	ϵ
	`ServletOutputStream.println(String)`	1	ϵ
	`jsp.JspWriter.print(String)`	1	ϵ
	`jsp.JspWriter.println(String)`	1	ϵ
HTTP response splitting	In class `javax.servlet.http.HttpServletResponse`:		
	`sendRedirect(String)`	1	ϵ
	`setHeader(String, String)`	1	ϵ
Command injection (stealth commanding)	In package `java.lang`:		
	`Runtime.exec(String, ...)`	1	ϵ
	`Runtime.exec(String[], ...)`	1	[]
	`System.load(String)`	1	ϵ
	`System.loadLibrary(String)`	1	ϵ
Path traversal	In package `java.io`:		
	`FileReader(String)`	1	ϵ
	`FileWriter(String)`	1	ϵ
	`FileInputStream(String)`	1	ϵ
	`FileOutputStream(String)`	1	ϵ
	`File(String)`	1	ϵ

Figure 17: Summary of sink descriptors for different vulnerability types.

Derivation descriptor = $\langle m, n_s, p_s, n_d, p_d \rangle$ Method m	n_s	p_s	n_d	p_d
`String(String)`	1	ϵ	-1	ϵ
`String(StringBuffer)`	1	ϵ	-1	ϵ
`String.toString()`	0	ϵ	-1	ϵ
`String.toLowerCase()`	0	ϵ	-1	ϵ
`String.toUpperCase()`	0	ϵ	-1	ϵ
`String.replace(char, char)`	0	ϵ	-1	ϵ
`String.replaceAll(String, String)`	0, 2	ϵ	-1	ϵ
`String.replaceFirst(String, String)`	0, 2	ϵ	-1	ϵ
`String.split(String)`	0	ϵ	-1	[]
`String.substring(int, ...)`	0	ϵ	-1	ϵ
`String.trim()`	0	ϵ	-1	ϵ
`String.concat(String)`	0, 1	ϵ	-1	ϵ
`StringBuffer(String)`	1	ϵ	-1	ϵ
`StringBuffer.toString()`	0	ϵ	-1	ϵ
`StringBuffer.append(String)`	0, 1	ϵ	-1	ϵ
`StringBuffer.append(StringBuffer)`	0, 1	ϵ	-1	ϵ
`StringBuffer.append(...)`	0	ϵ	-1	ϵ
`StringBuffer.delete(int, int)`	0	ϵ	-1	ϵ
`StringBuffer.deleteCharAt(int)`	0	ϵ	-1	ϵ
`StringBuffer.insert(int, String)`	0, 2	ϵ	-1	ϵ
`StringBuffer.insert(int, Object)`	0, 2	ϵ	-1	ϵ
`StringBuffer.insert(int, ...)`	0	ϵ	-1	ϵ
`StringBuffer.insert(int, char[] int, int)`	0	ϵ	-1	ϵ
`StringBuffer.replace(int, int, String)`	0, 3	ϵ	-1	ϵ
`StringBuffer.substring(int, ...)`	0	ϵ	-1	ϵ
`StringTokenizer(String, ...)`	0, 1	ϵ	-1	ϵ
`StringTokenizer.nextElement()`	0	ϵ	-1	ϵ
`StringTokenizer.nextToken()`	0	ϵ	-1	ϵ

Figure 18: Summary of derivation descriptors in Java APIs. We abbreviate the signatures by indicated immaterial parameters with "...". To save space, we also list all possible values of source argument number n_s separated by commas.