IMPROVING SOFTWARE SECURITY WITH PRECISE STATIC AND RUNTIME ANALYSIS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Benjamin Livshits

December, 2006

Abstract

The landscape of security vulnerabilities has changed dramatically in the last several years. While buffer overruns and format string violations accounted for a large fraction of all exploited vulnerabilities in the 1990s, the picture started to change in the first decade of the new millennium. As Web-based applications became more prominent, familiar buffer overruns are now far outnumbered by *Web application vulnerabilities* such as SQL injections and cross-site scripting attacks. These vulnerabilities have been responsible for a multitude of attacks against large e-commerce sites, financial institutions and other sites, leading to millions of dollars in damages.

In this thesis, we describe the Griffin project, which provides a comprehensive static and runtime compiler-based solution to a wide range of Web application vulnerabilities. Our approach targets large real-life Web-based Java applications. Given a vulnerability description a static code checker is generated. The checker statically analyzes the code and produces vulnerability warnings. Alternatively, a specially instrumented, secured version of the original application bytecode is produced, which can be deployed on a standard application server alongside other applications.

To make our approach to vulnerability detection and prevention both extensible and user-friendly, vulnerability specifications are expressed in PQL, a Program Query Language. The initial PQL vulnerability specification is provided by the user, but the majority of the specification can be shared among multiple applications being analyzed. This is because most of the PQL specification is specific to the framework being used, such as Java J2EE, rather than the application.

The static checker generated based on the PQL specification finds vulnerabilities by analyzing the Web-based applications. The static approach is sound, which ensures that it finds all vulnerabilities captured by the specification in the statically analyzed code. We evaluate analysis features such as context-, object- and map sensitivity that help keep the number of false positives low.

Modern Java applications make an extensive use of runtime reflection; this is especially common in extensible applications that support plugins and extensions. Being oblivious to reflection in the process of call graph construction leads to large portions of the application being ignored. This thesis describes the first call graph construction algorithm to explicitly address the issue of reflection. When reflection *is* taken into account by our approach, the size of the resulting application call graph often increases many-fold.

Conservative static analysis provides an over-approximation of runtime behavior. As such, it is impossible to have a conservative static solution that is completely free of false positives, despite our efforts to improve static analysis precision. Therefore, as an alternative to static analysis, secured application executables can be automatically generated based on the same PQL vulnerability specification. As an alternative to terminating the Web application whenever an exploit is detected, vulnerability recovery rules may be provided as part of the PQL specification. Recovery allows injecting security into existing applications without sacrificing application availability. Finally, we show how static analysis can be used to significantly reduce the instrumentation overhead required for runtime protection.

Our experimental results show that Griffin provides effective and practical tools for finding and preventing security vulnerabilities. We were able to find a total of 98 security errors, and all but one of our 11 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 static analysis reported false positives for only one of 11 applications we have analyzed. While the runtime overhead can be quite high for our runtime protection, information we compute statically allows us to reduce the number of necessary instrumentation points dramatically, reducing the dynamic overhead to below 10% in the majority of cases. Finally, our runtime system was able to recover from all exploits we performed against it in practice.

Acknowledgments

This thesis would not exist without the help, advice, and encouragement of many people. I thank my advisor, Monica, for giving me the freedom and resources to pursue research that interested and excited me. I learned a great deal from her about how to create the best research I can and how to persevere and not to give up when faced with adversity.

I am also indebted to several people in particular who greatly influenced my research approach style and style. Mooly Sagiv was kind enough to offer me a visiting researcher position at Tel Aviv University, which taught me about choosing good problems to attacks and led me to re-evaluate my research strategy. I am grateful to Alex Aiken for many thought-provoking conversations, which provided an often needed complementary view on my research, program analysis, and the overall field of computer research in general. I learned a great deal about how to pick good problems to work on from Dawson Engler; his research served as an example of how to stay grounded and how seek out problems with real-life visibility and impact while in academia. Finally, I want to thank Andreas Zeller, who was always supportive and upbeat and also encouraging of my work with one of his graduate students, Tom Zimmerman. Working with Tom made my foray into revision history mining both fun and productive [133, 134, 214].

I am grateful to the professors at Stanford who, while not being directly involved in my thesis work, nevertheless influenced me greatly. I want to thanks John Mitchell, who taught me much about teaching and research. His broad theoretical knowledge and a positive, laid-back attitude has always brought a breath of fresh air during the most challenging of times. I was particularly fortunate to TA a compiler course with

Jeff Ullman. While an honor in itself, it proved an invaluable experience. I learned a great deal from his approach to research and life in general. My thanks also go to Wei Li who provided many valuable insights based on his experiences in both the academia and the industry.

Many thanks go to the members of SUIF group at Stanford, who made the lab an exciting place to work, debate, and hang out. So many people enriched my experience. I am also grateful to the previous generation of students in the SUIF group, who taught me so much about systems research and also in terms of strategies for getting things done. I am particularly grateful to David Heine, Brian Murphy, Amer Divan, Michael Martin, Chris Unkel, Ramesh Chandra, and Costa Szapunkakis.

The material in Chapters 2 and 3 is based on "Finding Security Errors in Java Programs with Static Analysis" published in Usenix Security [129, 130]. John Whaley and especially Michael Martin contributed greatly to the material described in Chapters 4 and 5 and first presented in our joint APLAS and OOPSLA papers, entitled "Reflection Analysis for Java" [131, 132] and "Finding Application Errors and Security Vulnerabilities Using PQL: a Program Query Language" [140, 141], respectively. Runtime translation of PQL queries in Section 5.2 is part of Michael's contribution and is provided here for completeness reasons.

I am also indebted to John Whaley for building and supporting much of the compiler infrastructure that enabled a significant portion of the research described in this thesis. Other students in the Stanford Computer Science Department often provided a much welcome outlet for thought, worries, and complains. I am especially grateful to Ted Kremenek, Paul Thohey, and Iddo Lev. I also had the fortune of working with and mentoring many wonderful and hard-working master's students over the last several years, including Ben DeAngelo, Jean-Gabriel Morard, Chang-Seo Park, Will Robinson, and Naeim Semsarilar.

I am grateful to many people in the security community who encouraged this research and provided a great deal of help in its early stages. I am grateful to Brian Chess for turning my attention to the problems of Web application security in early 2004. I thank Amit Klein for reviewing some of my earlier writing and providing detailed clarifications on some of the more obscure security vulnerabilities.

Professor Dan Boneh at Stanford has also been a positive and encouraging influence on my research efforts.

This research was funded in part by the National Science Foundation under grants NSF CCF-0326227 and NSF/UCB CCF-0424422. I am also grateful to the NSF for awarding me with a multi-year Graduate Research Fellowship early in my graduate student career.

I devote a separate paragraph to the dissertation support group run by Ron Burg. I joined the group in late 2004, when my dissertation progress was slow, there was no particularly exciting thesis topic in sight, and the future looked especially bleak. In the group, I was fortunate to find a wonderful close-knit group of folks who were struggling with many of the same issues in their dissertation process. If not for the magic of group support over the subsequent two years, this thesis would most likely not have materialized. I wish you guys the very best!

For me, graduate school was tolerable because I had a group of close friends. Mine were the best: Mark Dilman, Sofya Pogreb, Elena Vileshina, and many others. Finally, I thank my family. To my parents, who were ultimately the people who prepared me for this endeavor. I owe you all my success.

Sonnet

All we need is fourteen lines, well, thirteen now,
and after this one just a dozen
to launch a little ship on love's storm-tossed seas,
then only ten more left like rows of beans.
How easily it goes unless you get Elizabethan
and insist the iambic bongos must be played
and rhymes positioned at the ends of lines,
one for every station of the cross.
But hang on here wile we make the turn
into the final six where all will be resolved,
where longing and heartache will find an end,
where Laura will tell Petrarch to put down his pen,
take off those crazy medieval tights,
blow out the lights, and come at last to bed.

— *Billy Collins*

Griffin Software Security Project

http://suif.stanford.edu/~livshits/work/griffin/

Contents

List of Figures

Chapter 1

Introduction

The security of Web applications has become increasingly important in the last decade. Web applications are rapidly becoming the norm for a wide range of software development projects, as client-server solutions are getting less popular. 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.

1.1 Introduction

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 [37]. 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 [200]. 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 [20, 70].

According to the 2005 E-Crime Watch survey conducted in cooperation with the United States Secret Service, 43% of respondents reported an increase in e-crimes and intrusions over the previous year [138]. Overall 70% of respondents reported at least one e-crime or intrusion was committed against their organization. During the first six months of 2005, malicious code that exposed confidential information represented 74% of the top 50 malicious code samples, according to Symantec's Internet Security Threat Report Volume VIII [40]. The report also documents 1,872 vulnerabilities in the first half of 2005, the most ever recorded since the inception of the report. Despite this sampling of data pointing to the increasing threat of directed attacks, the threat is likely still understated. Many directed attacks go unreported for the following reasons:

- Many organizations try to suppress the fact that they were attacked in the hope of avoiding negative publicity and damage to their reputation.
- Many organizations that have been attacked simply do not know that they have been the victim of a targeted attack.

While a great deal of attention over the last decade has been given to network-level attacks such as port scanning, about 75% of all attacks against Web servers target Web-based applications, according to a recent survey [89]. 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 [41, 177, 194]. 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), Struts, Web-Works, and Tapestry also helped to promote the adoption of Java as a language for implementing e-commerce applications such as Web stores, banking sites, customer information management 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 implement. However, despite Java language's safety, it is possible to make logical programming errors that lead to vulnerabilities such as SQL injections [6, 7, 59] and cross-site scripting attacks [33, 87, 179]. 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. Moreover, certain types of attacks may result in the attacker gaining complete control over the underlying system.

The fact that many applications are deployed on external sites greatly increases the perimeter of the attack. Many Web sites need to be made public in order to give the necessary access to their customers, however, this also exposes them to malicious hackers. A good example of this is a recent attack on a government Web site `ri.gov` perpetrated by a college student living in Eastern Europe [170]. The enabling mechanism for the attack was the presence of a SQL injection vulnerability, which allowed the hacker to discover the structure of database tables and then to execute a shell command through the underlying SQL server database. This incident led to the theft of hundreds of credit card numbers.

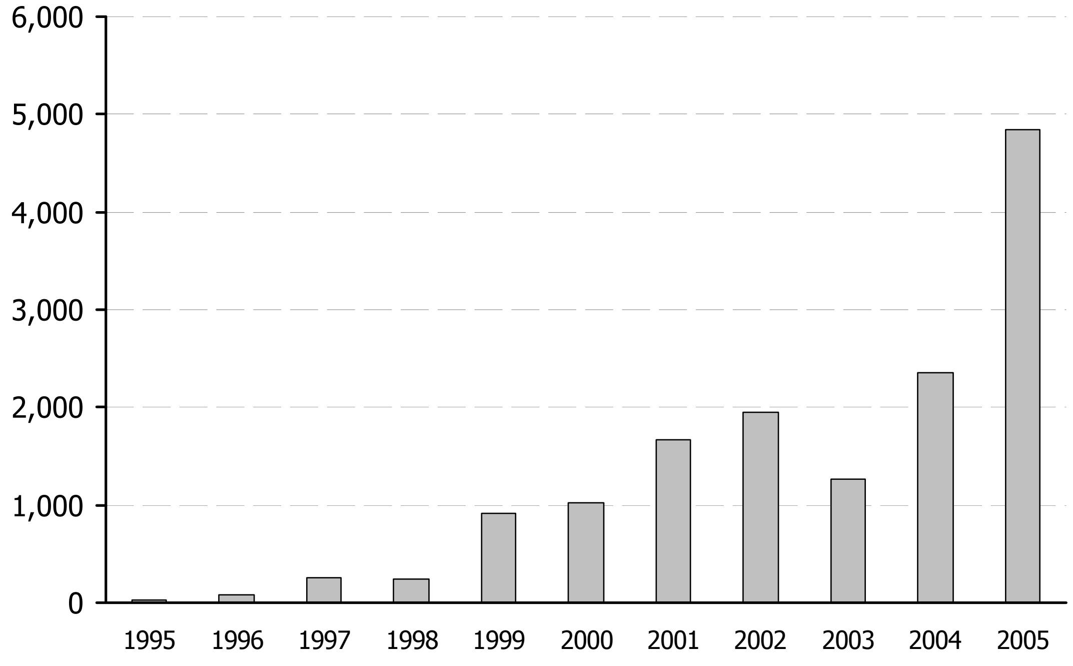


Figure 1.1: Number of vulnerabilities reported by year (based on NIST/DHS data).

1.2 Web Application Vulnerability Statistics

To further motivate our focus on Web application vulnerabilities in this thesis, this section presents statistics that demonstrate how common various categories of vulnerabilities are. While there are many cataloguing sites that collect vulnerability reports, reliable statistics on the frequency of different vulnerability categories is hard to come by. Below we report on the data obtained from some publicly available sources that shed light on this matter.

1.2.1 NIST Study

The National Institute of Standards and Technology (NIST) and the Department of Homeland Security have been aggregating vulnerability data for many years [156]. Statistics summarizing the total number of vulnerabilities in the NIST database are presented in Figure 1.1. While the numbers for 2006 are not yet available at the time

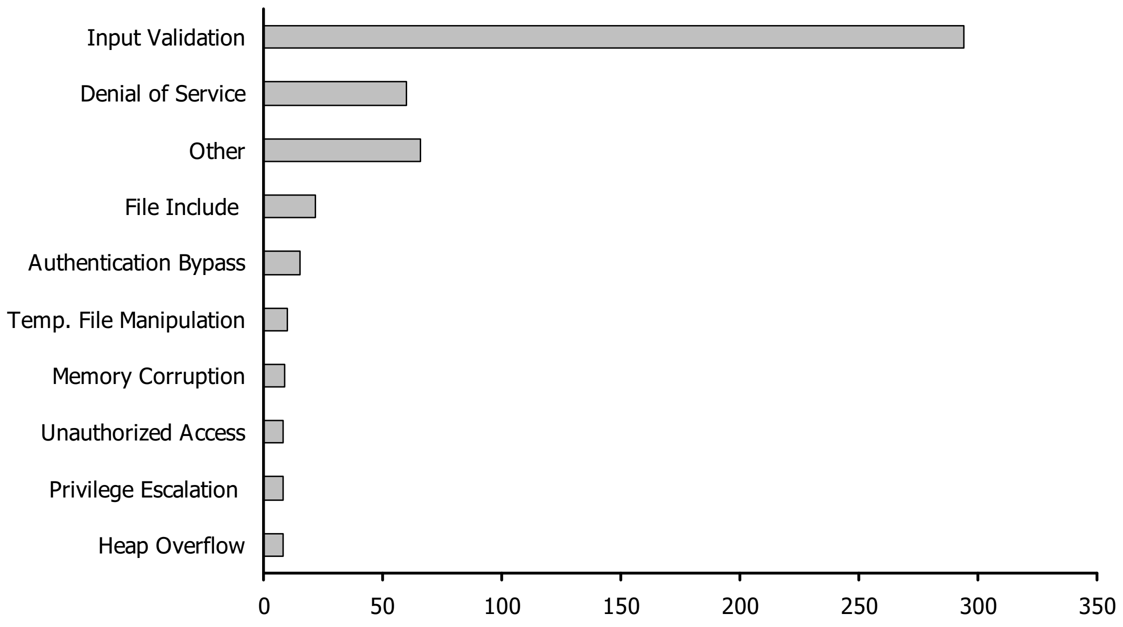


Figure 1.2: Relative frequency of vulnerabilities in the SecurityFocus.com sample.

of this writing, the number of vulnerabilities is projected to be higher than in 2005.

As can be seen from Figure 1.1, there is a slight decline in the number of vulnerabilities in 2003, followed by a sharp increase in 2004 and 2005. While it is difficult to validate these claims precisely, one interpretation of this anomaly is that the decline in 2003 is attributable to most of "shallow" buffer overruns being found. The sharp increase in 2004 can be attributed to Web application vulnerabilities becoming commonplace.

1.2.2 SecurityFocus.com Study

To gain insight into the relative frequencies of different vulnerability types, we considered a sample of 500 vulnerability reports obtained from the SecurityFocus.com database. This vulnerability sample spans a week in November 2005. The format of the vulnerability database allows for relatively easy processing and classification of this data. A coarse classification of these vulnerabilities is shown in Figure 1.2. It is apparent from the picture that input and output validation vulnerabilities account for over 50% of the sample.

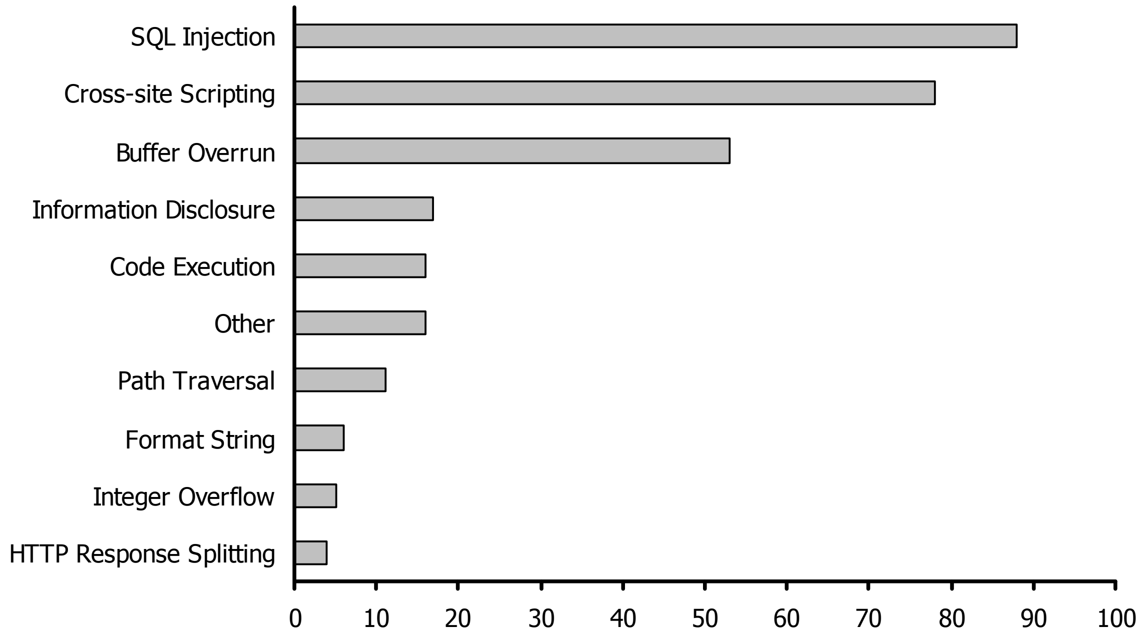


Figure 1.3: Relative frequency of input/output validation vulnerabilities.

Focusing in on input and output validation vulnerabilities shows that most are vulnerabilities specific to Web applications, as can be seen from Figure 1.3. In fact, the infamous buffer overrun issue comes in a distant third, trailing SQL injection and cross-site scripting attacks.

1.2.3 Relative Popularity of Java

The focus of our work is on large mission-critical software systems written in Java. The rise of such systems is in part due to the type safety and modularization features of Java, which allow the development of large systems, which we defined to be over 100,000 lines of code. The popularity of Java for Web applications has also been fueled by the rise of Web application development frameworks such as J2EE, Tapestry, WebWorks, Apache Struts and many others.

While no reliable data on this subject is available, anecdotal evidence suggest that Java J2EE applications comprise a significant fraction of Web applications at large financial firms and e-commerce sites [181]. .NET applications, which are typically

February	2000	Cross site scripting
October	2001	Path traversal
May	2001	SQL injection
February	2002	Cookie poisoning
March	2003	Cross site tracing
March	2004	HTTP response splitting
June	2005	HTTP request smuggling
July	2005	DOM-based cross-site scripting
June	2006	Domain contamination

Figure 1.4: History of common Web application vulnerabilities. The date of the first report describing a vulnerability found "in the wild" or the article describing the vulnerability is given.

written in C#, account for another big fraction of Web applications. Most of the ideas described in this thesis apply to applications written in C# as well. One C# language feature potentially complicating the pointer analysis described in Chapter 3 is the presence of stack-allocated objects. Moreover, the presence of unmanaged code makes the soundness guarantees of our static approach much harder to sustain; see Section 3.6 for a discussion of soundness.

As the recent research in vulnerability detection in scripting languages such as PHP demonstrates [95, 207], a different approach is typically required for that class of languages. In particular, control flow of the program is typically more important than it is in Java. Furthermore, the presence of the `eval` construct makes soundness very difficult to maintain.

1.2.4 History of Web Application Vulnerabilities

Web application vulnerabilities have a short but an illustrious history. Figure 1.4 shows the first time each type of vulnerability was discovered. In many cases, vulnerabilities were first discovered and described in a paper by a vulnerability researcher, with actual exploits following soon thereafter.

We can also expect new types of vulnerabilities to appear in the future. For instance, two new flavors of injection attacks, *control injection* and *reflection injection*, which allow the hacker to influence the control flow path taken by an application and what classes are created reflectively are entirely possible [178]. How these vulnerabilities can be exploited remains to be seen.

1.3 Overview of Web Application Vulnerabilities

As supported by the statistics presented in Section 1.2.2, of all vulnerabilities identified in Web applications, problems caused by *input/output validation* are recognized as being the most common. These vulnerabilities, also sometimes referred to as *taint-style* or *information flow* vulnerabilities are the focus of this thesis.

Our statistics in Section 1.2.2 are further supported by security surveys. According to an influential survey performed by the Open Web Application Security Project [157], unvalidated input is the number one security problem in Web applications. Many such security vulnerabilities have recently appeared on specialized vulnerability tracking sites such as `SecurityFocus` and are widely publicized in the technical press [151, 157]. Recent reports include SQL injections in Oracle products [122] and cross-site scripting vulnerabilities in Mozilla Firefox [112].

Input validation problems are similar to those handled dynamically by the *taint mode* in Perl [197]. Perl maintains a runtime taint bit it attaches to every value. The taint bit is set for data that comes from the outside (return result of file or network read methods, etc.). Perl automatically un-taints a value that is matched against a regular expression. While a good heuristic, this approach is obviously not safe if the regular expression is `/(.*)/`, which matches all strings. Moreover, when the blacklist validation approach is used, regular expressions are created to recognize malicious values, in which case matching strings should definitely *not* be untainted. Our approach to controlling the flow of taint is considerably more extensible, as summarized in Section 2.1.1.

In Section 1.3.1 below we list the vulnerabilities this thesis addresses. However, the framework presented here is general enough to encompass other, yet undiscovered vulnerability types. As mentioned above, reflection injection refers to the user being able to control which classes are reflectively instantiated at runtime [178]. While we are able to find such vulnerabilities, none has been observed in the wild yet.

1.3.1 Categorization of Taint-Style Vulnerabilities

To exploit unchecked input, an attacker must achieve two goals:

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

- **Parameter manipulation**: pass specially crafted malicious values in fields of HTML forms.
- **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.
- **Non-Web sources**: set command-line parameters to malicious values.

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.

```
javax.sql.Connection con  = ...
javax.servlet.http.HttpServletRequest request = ...;
String userName = request.getParameter("username");
String query = "SELECT * FROM Users " + " WHERE name = '" + userName + "'";
con.execute(query);
```

Figure 1.5: Simple SQL injection example.

The kinds of vulnerabilities described above are widespread in today's Web applications. A recent empirical study found that parameter manipulation, SQL injection, and cross-site scripting attacks account for more than a third of all reported Web application vulnerabilities [185]. While different on the surface, all types of attacks listed above are made possible by user input that has not been (properly) validated.

In this chapter we only outline the basics of each vulnerability. More detailed information about the vulnerabilities outlined below can be found in "The 21 Primary Classes of Web Application Threats" [151] and the "OWASP Secure Development Guide" [158].

1.3.2 SQL Injection Example

Let us start with a discussion of SQL injections, one of the most well-known security vulnerabilities found in Web applications. SQL injections are caused by unchecked user input being passed to a back-end database for execution [6, 7, 59, 109, 123, 180]. The hacker may embed SQL commands into the data he sends to the application, leading to unintended actions performed on the back-end database. When exploited, a SQL injection may cause unauthorized access to sensitive data, updates or deletions from the database, and even shell command execution.

Example 1.1. A simple example of a SQL injection is shown in Figure 1.5. This code snippet obtains a user name (`userName`) by invoking method `request.getParameter("username")` 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 in the Oracle dialect of SQL to indicate the beginning of a comment. (Other databases use a slightly different syntax.) Therefore, the `WHERE` clause of the query effectively becomes the tautology `name = '' OR 1 = 1`. Thus, the attacker may circumvent the name check and gain 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 steps:

Step 1. injecting malicious data into the application, and

Step 2. using the data to manipulate the application.

The former corresponds to the *sources* of a tainted object propagation problem and the latter to the *sinks*. The rest of this section presents attack techniques and examples of how exploits may be created in practice[1].

1.3.3 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 is 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

[1]Most security exploitation techniques described in this chapter are independent of the programming language being used. To make the discussion below concrete, we use Java examples. Code snippets we show in this section are extracted from a suite of Web application benchmarks described in Chapter 6.

easy to mount: an attacker needs little more than a standard Web browser to attack Web applications in most cases.

Parameter Manipulation

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.

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

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

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

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

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

may in fact increase the account balance. □

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

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

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

Hidden Field Manipulation

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

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

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

HTTP Header Manipulation

HTTP headers typically remain invisible to the user and are used only by the browser and the Web server. However, some Web applications do process these headers, and attackers can inject malicious data into applications through them. While a normal Web browser will not allow forging the outgoing headers, multiple freely available tools allow a hacker to craft an HTTP request leading to an exploit [35].

Example 1.3. An HTTP request fragment is shown in Figure 1.6. 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

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

Figure 1.6: An HTTP request fragment.

pass it to a database to look up a language-specific text message. If 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 the name of file that contains messages for the correct language, an attacker may be able to launch a path-traversal attack [158]. □

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 [158].

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 [103]. 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.

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

(a)

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

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

(b)

Figure 1.7: 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`.

Example 1.4. Consider the HTTP `GET` request in Figure 1.8. 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 than 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 to 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 [18].

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,

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

Figure 1.8: An HTTP `GET` request containing a cookie.

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.

1.3.4 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.

In this section we present the following techniques: SQL injections, cross-site scripting vulnerabilities, HTTP response splitting attacks, path-traversal attacks, and command injections.

SQL Injections

SQL injections (first described in Section 1.3.2) are caused by unchecked user input being passed to a back-end database for execution. When exploited, a SQL injection may cause a variety of consequences from leaking the structure of the back-end database to adding new users, mailing passwords to the hacker, or even executing arbitrary shell commands.

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

Example 1.5. Figure 1.7 shows two ways to update the salary of an employee, whose id is provided. The first method in Figure 1.7 (a) uses string concatenation to construct the query and leads to potential SQL injection attacks; the second in Figure 1.7 (b) uses `PreparedStatements` and is safe from SQL injection attacks. □

Most SQL injections we have encountered can be categorized as the result of constructing SQL statements directly instead of using `PreparedStatements`. 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 may 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`, e.g. using non-constant template strings for constructing prepared statements defeats the purpose of using them in the first place.

Cross-site Scripting Vulnerabilities

Cross-site scripting occurs when dynamically generated Web pages display input that has not been properly validated [33, 38, 87, 102, 179]. An attacker may embed malicious JavaScript code into dynamically generated pages of trusted sites. When executed on the machine of a user who views the page, these scripts may hijack the user account credentials, change user settings, steal cookies, or insert unwanted content (such as ads) into the page. At the application level, echoing the application input back to the browser verbatim enables cross-site scripting.

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

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

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

HTTP Response Splitting

HTTP response splitting is a general technique that enables various new attacks including Web cache poisoning, cross-user defacement, sensitive page hijacking, as well as cross-site scripting [104]. 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 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 from one of our benchmark applications, `personalblog`:

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

Path Traversal

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

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

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

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

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

Command Injection

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

1.3.5 Unvalidated Output and Information Leaks

While the focus of the discussion above has been on poorly validated user input, many of the same concerns apply to validating the *output* that is passed from the Web application to the user's browser.

A common Web application vulnerability that stems from poorly validated output is *information leaks*. For example, when a piece of sensitive data is leaked from a back-end underlying database, this might be a sign of trouble. Another common error pattern is revealing too much in exception messages. When a Web application encounters an exception, these messages containing potentially sensitive information are often sent to the browser. Furthermore, cross-site scripting attacks can be seen as unvalidated output issues.

Conceptually, unvalidated output attacks are really the same as unvalidated input attacks and they fit equally well into our analysis framework described in Chapter 2. However, fewer potential output validation issues are clear-cut cases of exploitable vulnerabilities. Often, leaking a piece of data from the database does not have any undesirable consequences. Because of this, we chose not to look for output validation issues in our experiments. However, if needed, these types of vulnerabilities can be easily expressed in the Griffin framework.

1.4 Existing Solutions

In this section we briefly summarize existing approaches to Web application security issues. In this section, our focus is on accepted industry practices. We postpone a detailed discussion of related research until Chapter 7.

1.4.1 Client-side Protection

The simplest approach used to combat Web application security issues is to perform validation on the client side. This is typically done by having Javascript execute in the client's browser when a form is submitted. While a common technique, client-side protection is quite easy to circumvent:

- An attacker can save the Web page in the browser, edit it to remove the validating Javascript code, and resubmit the form again.
- Another technique consists of crafting HTTP packets manually or with the help of tools such as PacketCrafter [108]. Specially crafted malicious packets can be sent to the application directly, completely circumventing browser-based protection.

While more sophisticated browser-based security measures have recently been proposed [100], clearly, simple client-side validation is completely helpless when either of the two approaches described above is used.

1.4.2 Penetration Testing

Penetration testing is a commonly used technique for improving Web application security [105, 135, 136, 187]. Penetration testing is an approach that involves a person or a tool constructing potentially malicious inputs and feeding them to the Web application, mimicking the actions of a hacker. This is done in an effort to either crash or exploit the application being tested.

The most common difference between penetration testing strategies is the amount of knowledge about the implementation of the system being tested. Black box testing assumes no prior knowledge of the infrastructure to be tested. At the other end of the spectrum, white box testing provides the testers with complete knowledge of the infrastructure to be tested, often including network diagrams, source code, and IP addressing information. There are also several variations in between, often referred to as gray box testing.

The relative merits of these approaches are debatable. It is argued that black box testing most closely simulates the actions of a real hacker, however this ignores the fact that any targeted attack on a system most probably requires some knowledge of the system, and any insider attacker would be in possession of as much information as the system owners. In most cases it is preferable to assume a worst-case scenario and provide the testers with as much information as they require, assuming that any determined attacker would already have acquired this. The end-goal of penetration testing is to either produce a crash or to get the application to disclose sensitive data.

The difficulty of performing such tests manually lead to the development of automatic techniques such as Cenzic's Hailstorm [32]. However, no matter how automated the approach, as a testing technique, penetration testing still suffers from lack of coverage and therefore likely misses a large fraction of vulnerabilities.

1.4.3 Application Firewalls

According to the Web Application Security Consortium Glossary [199], an application firewall is "An intermediary device, sitting between a Web client and a Web server, analyzing layer-7 messages for violations in the programmed security policy. A Web application firewall is used as a security device protecting the Web server from attack." Standard firewalls are designed to restrict access to certain ports, or services that an administrator does not want unauthorized people to access. Application firewalls are often called "deep packet inspection firewalls" because they examine every request and response within the HTTP, SOAP, XML-RPC, and Web service layers.

Application firewalls vary in the filtering approach, which includes both whitelisting and blacklisting or a combination of the two. The blacklisting approach consists of maintaining a database of attack signatures, not unlike regular firewalls or intrusion detection software. The whitelisting relies on having a model of "normal" traffic between the application and the client. Moreover, some application firewalls include the ability to fully configure filtering policies and learn them based on past traffic. However, coming up with proper policies that reduce the number of both false positives and false negatives is a formidable challenge.

Web application firewalls can be either software, or hardware appliance based and are installed in front of a Web or application server in an effort to try and shield it from incoming attacks.

1.4.4 Code Auditing for Security

Code reviews or code audits have been commonly used to improve the security posture of applications, leading to the creation of a large number of consulting companies and code auditing guidelines. While an effective approach to finding vulnerabilities, code auditing is a heavily manual task. Code auditing suffers from the following shortcomings:

- **Security vulnerabilities elude detection**. Vulnerabilities described in Section 1.3 involve following data flow in the program, which is especially difficult when a given piece of data is deposited into data structures. The branching factor adds another level of complication to the code auditing task: if a given method is invoked from dozens of different call sites, all of them need to be considered by the reviewer. While shallow security bugs are easily detected with code review, vulnerabilities that span a large number of methods, files, or directories are much less likely to be found with manual inspection. As our experiments described in Chapter 6 suggest, a significant number of vulnerabilities require examining numerous methods and files.

- **Lack of guarantees about the results.** A security audit typically improves the application security posture and is often the only approach to improving to overall application architecture. However, completeness is difficult to achieve: vulnerabilities are missed by manual efforts. This is especially true when it comes to vulnerabilities that span multiple methods and files, which are not uncommon, according to our experimental results.

- **Difficult to maintain continuous security.** The rapid pace of change in software development, especially when applied to Web applications, means that the security assessment may no longer be valid whenever the application code is changed. Since security code reviews are often done by outside consultants, new code changes may not get reviewed until the next consulting engagement, leaving the application in an insecure state for long periods of time.

1.4.5 Griffin Project Scope and Goals

The overarching goal of the Griffin project is to address a large range of Web applications security issues. Existing approaches described in Section 1.4 provide a best-effort attempt at improving Web application security. Instead, the goal of our research is to provide *guarantees* about the security posture of a given application.

The focus of the Griffin project is on large applications written in Java. As discussed in Section 1.2.3, Java is the primary development language for a large number of Web applications, especially mission-critical ones, making it an attractive analysis target.

1.5 Overview of Our Solution

The Griffin project aims to provide a comprehensive solution to a wide range of Web application security vulnerabilities. It offers a combination of static and runtime analysis techniques to achieve this goal.

The user of the Griffin project tools specifies what constitutes a vulnerability. Specifications are expressed in PQL, a Program Query Language [140]. PQL is a generic language that can be used to capture events that happen to objects, such as specific method calls being invoked with an object passed as a parameter or returned from a method. While PQL has been used to express a variety of queries for purposes ranging from debugging to finding optimization opportunities, in this thesis it is used to capture vulnerability queries.

Since most portions of vulnerability specification consist of J2EE library methods, and since the J2EE library is shared among most Java Web applications, the per-application specification effort in usually minor. Moreover, most vulnerabilities

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

    con.execute(param);
}
```

Figure 1.9: A very simple PQL query for finding SQL injections.

can be found with a "generic" specification that is specific to the Web application development framework such as J2EE or Apache Struts, which completely removes the need for user involvement. A very simple PQL query that captures only *some* SQL injection vulnerabilities is shown in Figure 1.9; more complete vulnerability queries are described in Chapter 2. This PQL query will locate all objects `param` which are returned from a call to `getParameter` and are passed into method `executeQuery`.

Based on the PQL vulnerability specification, a static analyzer is automatically generated. The static analysis runs over the application bytecode and produces vulnerability warnings. The static approach has the following important benefits:

- **Finds vulnerabilities early in the development cycle.** It is well-known that finding defects before they make it into a production system has great benefits because the communications between the party that detects the bug, the developer who fixes it, etc. are completely obviated [195]. Moreover, the window of opportunity that a vulnerable program presents is never opened with static assurance.

- **Explores all possible program executions.** Designing a test suite that adequately explores many program execution paths is generally a challenge. In the case of security exploits that often take trained hackers hours and days to design, adequate testing is an even harder problem. Static analysis obviates the need for security testing, while providing full path coverage.

- **No false negatives.** Soundness of our technique is one of the features that set it apart from other security efforts [88, 207]. Using a sound technique is the only way to provide *guarantees* about the security posture of a given application, as discussed in Section 3.6.

A significant disadvantage of a static analyzer is that it may suffer from imprecision. While a great deal of thinking and experimentation went into the design of our static analysis abstraction, the problem of soundly and precisely identifying security violations is undecidable. This means that in the worst case, false positives will still be reported no matter how precise we make our analysis technique.

The potential for imprecision in the static solution is one of the primary reasons we chose to provide a runtime alternative. Moreover, since a static analyzer would typically run as part of the development phase after the code is written, a runtime analysis is also a good fit in less established development environments, which may not have a separate testing phase or when the pace of development and deployment is especially fast. Finally, unlike a static technique, runtime analysis is not complicated by dynamic language features (such as reflection) and inter-language interpretability complications (such as `native` methods).

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

- **Keeps vulnerabilities from doing harm.** As discussed earlier, runtime analysis may be used in situations where the user is unwilling to consider the false positives. It also applies when the source code is unavailable or cannot be changed. The runtime technique is of great practical value in stopping existing vulnerabilities from being exploited. For example, an application that has an output validation vulnerability that may lead to an information leak can be terminated before the leak actually occurs.

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

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

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

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

1.5.1 Summary of Thesis Contributions

This section summarizes the contributions of the Griffin project.

- **Effective solution to an important practical problem.** This thesis describes what we believe to be the first effective and comprehensive solution to the problem of Web application security. It compares favorably with commonly used approaches such as client-side validation, penetration testing, and application firewalls. The problem of Web application security is only going to get bigger, as more and more of the infrastructure around us relies on Web applications.

- **A unified static and runtime analysis framework.** The Griffin analysis framework unifies multiple, seemingly diverse, recently discovered categories of Web application security vulnerabilities and proposes a combination of static and dynamic analysis techniques to detect and prevent them in large modern Java J2EE applications. The user can specify vulnerabilities to be found by the analysis in PQL, an intuitive and expressive language has been used to capture a wide range of properties that involve events applied to objects.

- **A powerful static analysis technique.** Our tool is the first practical static security analysis that utilizes a pointer analysis recently developed by Whaley and Lam [205]. We improve on the state of the art in pointer analysis by enhancing the object-naming scheme as well as the handling of maps [145, 146]. The precision of the analysis is effective in reducing the number of false positives issued by our tool. Our static technique also addresses a range of practical challenges that exist in analyzing Web applications, such as how to construct the static call graph without the need to analyze the application server.

- **Reflection analysis for Java.** We propose the first call graph construction algorithm for Java in the presence of reflection. We formulate a set of natural assumptions that hold in most Java applications and make the use of reflection amenable to static analysis. We propose a call graph construction algorithm that uses points-to information about strings used in reflective calls to statically find potential call targets. When reflective calls cannot be fully "resolved" at compile time, our algorithms determines a set of specification points — places in the program that require user-provided specification to resolve reflective calls. As an alternative to having to provide a reflection specification, we propose an algorithm that uses information about type casts in the program to statically approximate potential targets of reflective calls.

  We provide an extensive experimental evaluation of our analysis approach based on points-to results by applying it to a suite of six large open-source Java applications consisting of more than 600,000 lines of code combined. The conservative call graph obtained with the help of a user-provided specification results is a call graph than is almost 7 times as big as the original.

- **Runtime analysis technique.** The runtime technique is the first security prevention technique for Web application attacks of its kind. It relies on precise taint tracking to deliver a precise and sound prevention from Web application exploits. Another contribution is the design of a recovery technique, which allows vulnerable applications to continue running.

- **Experimental validation.** We present a detailed experimental evaluation of our system and the static analysis approach on a set of 11 large, widely-used open-source Java applications. To measure the effect of different analysis features, we also apply our techniques to a suite of over 100 small benchmarks.

  We found a total of 98 security errors, including two important vulnerabilities in widely-used libraries. Most of our benchmark applications had at least one vulnerability, and our analysis produced only 147 false positives, all of which were concentrated within a single benchmark.

  We also created exploits for some of the statically detected vulnerabilities. All the exploits were foiled in the dynamically protected applications. While the runtime overhead with the default instrumentation we apply is acceptable, averaging about 60%, when static results are used to reduce the number of instrumentation points, the overhead drops to below 10% in most cases.

1.6 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 outlines the scope of the Griffin project and presents our framework for analyzing Web application vulnerabilities. Chapter 3 focuses on the static analysis portion of the Griffin project. Chapter 4 describes our analysis of reflective constructs in Java. Chapter 5 discussed our runtime analysis approach. Chapter 6 summarizes our static and runtime analysis results and discusses our findings. Chapter 7 describes related work. Finally, Chapter 8 summarizes the contributions of this thesis and outlines some future research directions.

Chapter 2

Analysis Framework for Web Application Vulnerabilities

This chapter gives an overview of the Griffin analysis framework. Further information on static and runtime analysis can be found in Chapters 3 — 5.

2.1 Framework Overview

We start our discussion of the Griffin project architecture by focusing on the SQL injection example in Section 1.3.2. Conceptually, a vulnerability occurs because there is uninterrupted flow between a tainted object (as exemplified by String `userName` on line 3 in Figure 1.5) and a sink (`execute` on line 5). It is important to point out that in Java every string is a separate object. Moreover, a `String` object is immutable, meaning that once it becomes tainted, it will always remain so.

A vulnerability trace is a sequence of objects, such that every object is derived from the previous one, leading to a sink. Notice that the objects involved in a vulnerability trace are strings, represented in Java by standard library types `String`, `StringBuffer`, `StringBuilder`, `StringTokenizer`, etc. declared in package `java.lang`.

The overall goal of both static and runtime analyses is to locate such traces. While the example in Section 1.3.2 is quite simple, the trace is in fact 3 objects long:

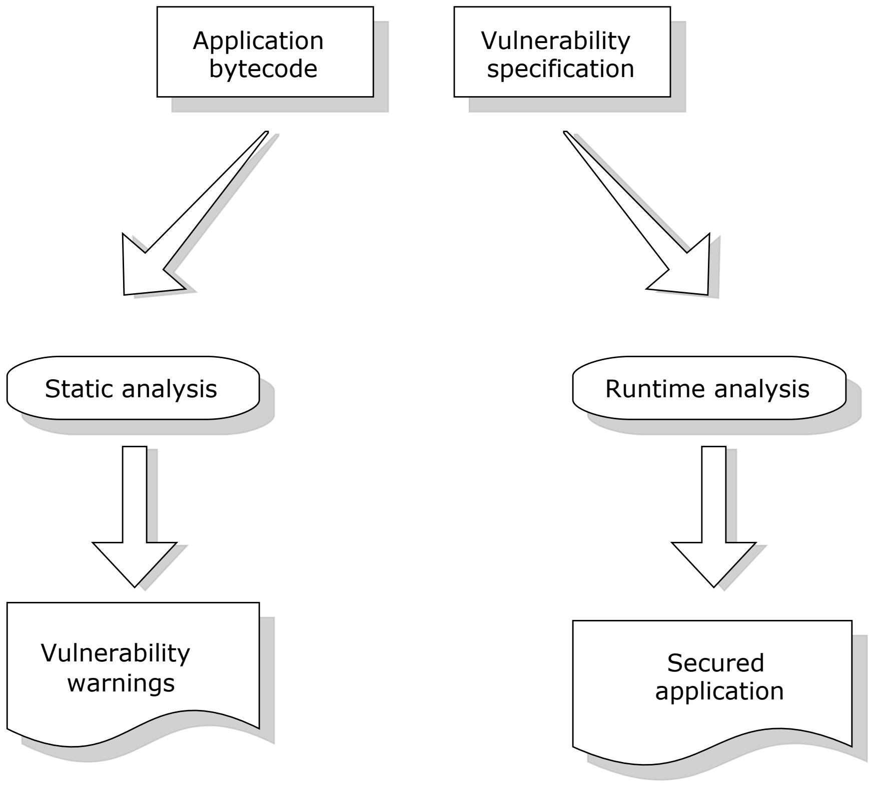


Figure 2.1: High-level architecture of the Griffin project.

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

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

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

2.1.1 Tainted Object Propagation Problem

In this section we formalize the tainted object propagation problem first described in Section 4.2. We start by defining the terminology that was first informally introduced in Example 1.

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

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

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

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

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

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

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

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

2.1.2 Derivation and Sanitization Descriptors

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

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

Unfortunately, there are numerous ways to obtain tainted objects from string objects in Java. Data contained in a string object propagates to any object derived

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

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

from the string through string concatenation, substring extraction, and other similar routines. For instance, `s.toLowerCase`() is derived from string `s`. Similarly, the result of `s` + ";" is derived from string `s`. Finally, `newStringTokenizer`(`s`) is derived from `s`, because the `StringTokenizer` object constructed out of a tainted string will produces potentially tainted tokens.

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

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

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

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

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

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

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

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

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

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

2.1.3 Security Violations

Below we formally define a security violation:

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

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

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

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

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

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

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

2.1.4 Specifications Completeness

If a specification is incomplete, important errors will be missed even if we use a sound analysis that finds all vulnerabilities matching a specification. Therefore, the problem of obtaining a complete specification for a tainted object propagation problem is an important one. However, it is hardly a unique issue for program analysis, as many other projects require a specification to be provided [9, 80, 194].

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

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

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

2.2 Specifying Vulnerabilities in PQL

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

For example, for static analysis, questions about static program properties may be expressed as Datalog queries [203] or type inference rules [107]. Datalog exposes the program intermediate representation (IR) as a set of relations. To determine static program properties, the user can subsequently query these relations. While giving the user complete control, Datalog queries expose too much of the program's internal representation to be practical for the casual use who does not want to learn the intricacies of the IR. The same argument applies to requiring the user to write runtime instrumentation code, leading to the development of numerous aspect-oriented systems such as AspectJ, etc. that make common tasks easier to accomplish [50, 99].

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

To make them accessible to developers, PQL queries are written in a familiar Java-like syntax. PQL serves as a layer of abstraction and, as a result, the user is not

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

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

    con.execute(derived);
}
```

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

required to become familiar with the details of static program internal representation or the internals of an instrumentation framework. Moreover, as shown in the Griffin project architecture in Figure 2.1, the same PQL query is used for both the static and the runtime portions of our analysis.

In this thesis, we only use a relatively limited and stylized form of PQL queries to formulate tainted object propagation problems; a more extensive description of PQL is found elsewhere [140]. Figure 2.4 provides a BNF grammar for PQL queries for reference purposes. Translation of tainted object propagation queries from PQL into static checkers and runtime instrumentation is described in more detail in Chapters 3 and 5, respectively.

2.2.1 Simple SQL Injection Query

Example 2.2. Figure 2.3 shows a PQL query for the SQL injection vulnerability in Example 1. It is important to point out and this is a relatively simple query example given here for the purpose of illustration that only addresses a small subset of all SQL injections that includes the code snippet in Figure 1.5. Queries capturing a wider range of vulnerabilities are discussed in Section 2.2.2.

```
queries      ⟶  query*
query        ⟶  query qid ( [decl [, decl]*] )
                 [returns declList ; ]
                 [uses declList ; ]
                 [matches { seqStmt }]
                 [replaces primStmt with methodInvoc ;]*
                 [executes methodInvoc [, methodInvoc]* ;]*
methodInvoc  ⟶  methodName(idList)
decl         ⟶  object [!] typeName id |
                 member namePattern id
declList     ⟶  object [!] typeName id ( , id )*|
                 member namePattern id ( , id )*
stmt         ⟶  primStmt | ∼ primStmt |
                 unifyStmt | { seqStmt }
primStmt     ⟶  fieldAccess = id |
                 id = fieldAccess |
                 id [ ] = id |
                 id = id [ ] |
                 id = methodName ( idList ) |
                 id = new  typeName ( idList )
unifyStmt    ⟶  id := id
                 ( [idList] ) := qid ( idList )
seqStmt      ⟶  ( commaStmt ; )*
commaStmt    ⟶  altStmt ( , altStmt )*
altStmt      ⟶  stmt ( "|" stmt )*
typeName     ⟶  id ( . id )*
idList       ⟶  [ id ( , id )* ]
fieldAccess  ⟶  id . id
methodName   ⟶  typeName . id
qid          ⟶  [A-Za-z_][0-9A-Za-z_ ]*
qid          ⟶  [A-Za-z_][0-9A-Za-z_ ]*
namePattern  ⟶  [A-Za-z*_ ][0-9A-Za-z*_ ]*
```

Figure 2.4: BNF grammar specification for PQL.

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

Figure 2.5: Main query for finding source-sink pairs.

Query `simpleSQLInjection` is described in more detail below. The **uses** clause of a PQL query declares all objects used 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 by the Java compiler for the string concatenation operation in Example 1. □

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

Figure 2.6: Transitive derived relation `derived`*.

2.2.2 Queries for a Taint Propagation Problem

In this section we describe how generic tainted object propagation queries are formulated. There is a direct correspondence between source, sink, derivation, and sanitization descriptors used in the problem (definition 2.1.2) and parts of the PQL query shown in Figure 2.5.

Generic Taint Propagation Queries

Query `main` shown in Figure 2.5 computes source-sink object pairs corresponding to static or runtime security violations for a given tainted object propagation problem. Intuitively, query `main` matches pairs of objects, such that the first object comes from a source, the second goes into a sink, and the second object is derived from the first one using zero or more derivation steps. The source and sink objects are denoted in the query as `sourceObj` and `sinkObj`, respectively. Events separated by semicolons in query `main` must occur in order, but can be separated by other events (such as method calls, etc.).

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

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

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

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

Figure 2.7: PQL subqueries for finding SQL injections.

Subquery `derived`$*$ shown in Figure 2.6 defines a transitive derived relation: object `y` is transitively derived from object `x` by applying subquery `derived` zero or more times. This query takes advantage of PQL's subquery mechanism to define a transitive closure recursively.

Instantiating Taint Propagation Queries

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

Example 2.3. This example describes subqueries `source`, `sink`, and `derived` shown in Figure 2.7 that can be used to match SQL injections, such as the one described in Example 1. Usually these subqueries are structured as a series of alternatives separated by |. The wildcard character _ is used instead of a variable name if the identity of the object to be matched is irrelevant.

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

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

2.3 Static Analysis Overview

A tainted object propagation problem expressed in the form of a PQL query is automatically translated into a query in Datalog, a logic programming language [192]. Information about the input program is summarized as a set of database relations that are represented in a highly compressed form in bddbddb, a BDD-based deductive database [205]. Datalog queries are combined with the results of pointer analysis as well as the initial program relations and are fed to the `bddbddb` solver to obtain the final solution. According to the PQL query formulation in Figure 2.5, only static approximations of the source and sink objects are returned.

Recovering the full vulnerability trace requires simple post-processing of analysis results. The analysis for finding the relevant instrumentation points is discussed further in Section 3.5.4.

2.4 Runtime Analysis Overview

The second component of the Griffin project is runtime analysis. There are several benefits of runtime analysis in this context.

- Even the most precise conservative static analysis will suffer from false positives in the worst case. If the user does not want to deal with false positives generated from the static checker, runtime analysis is a good alternative. Runtime analysis monitors application execution, only flagging dangerous vulnerability patterns if and when they occur at runtime.

- While some organizations have an established software development process, which includes well-defined development and testing phases, others do not. As a result, requiring developers to adopt a static analysis tool is not a viable option at those organizations. However, a runtime tool does not require much involvement on developers' part. A PQL specification can often be defined by the product architect or a system administrator.

Taint propagation problems expressed as PQL queries are automatically translated into finite state machines runtime that run alongside the application as it executes. If a query match is detected, the application either terminates, thus preventing a potential exploit, or recovers from the vulnerability, if recovery code is provided as part of the specification. More information about vulnerability recovery is provided in Section 5.5.

2.5 Chapter Summary

In this chapter we have presented the architecture of the Griffin project, focusing on the user of PQL to specify vulnerabilities of interest. Chapters 3 and 5 cover the details of static and runtime analysis, respectively.

Chapter 3

Static Analysis

This chapter describes the static analysis techniques used in the Griffin project. Static analysis results are summarized in Chapter 6.

3.1 Chapter Overview

Static analysis techniques described in this chapter represent the bulk of this thesis's contribution. A diagram representing the static portion of the Griffin project is shown in Figure 3.1. The chapter is organized as follows. We start with a code auditing example that motivates our analysis technique in Section 3.2. Section 3.3 summarizes our static analysis approach and lays the ground for the rest of the chapter. Section 3.4 describes our pointer analysis precision enhancements. Section 3.5 formalizes the notion of a static security vulnerability and explains how it relates to pointer analysis results; it also describes how a static analyzer (or checker) is automatically generated from a PQL vulnerability specification. Section 3.6 addresses the issue of analysis soundness. Section 3.7 describes our approach to increasing the static coverage of our analysis.

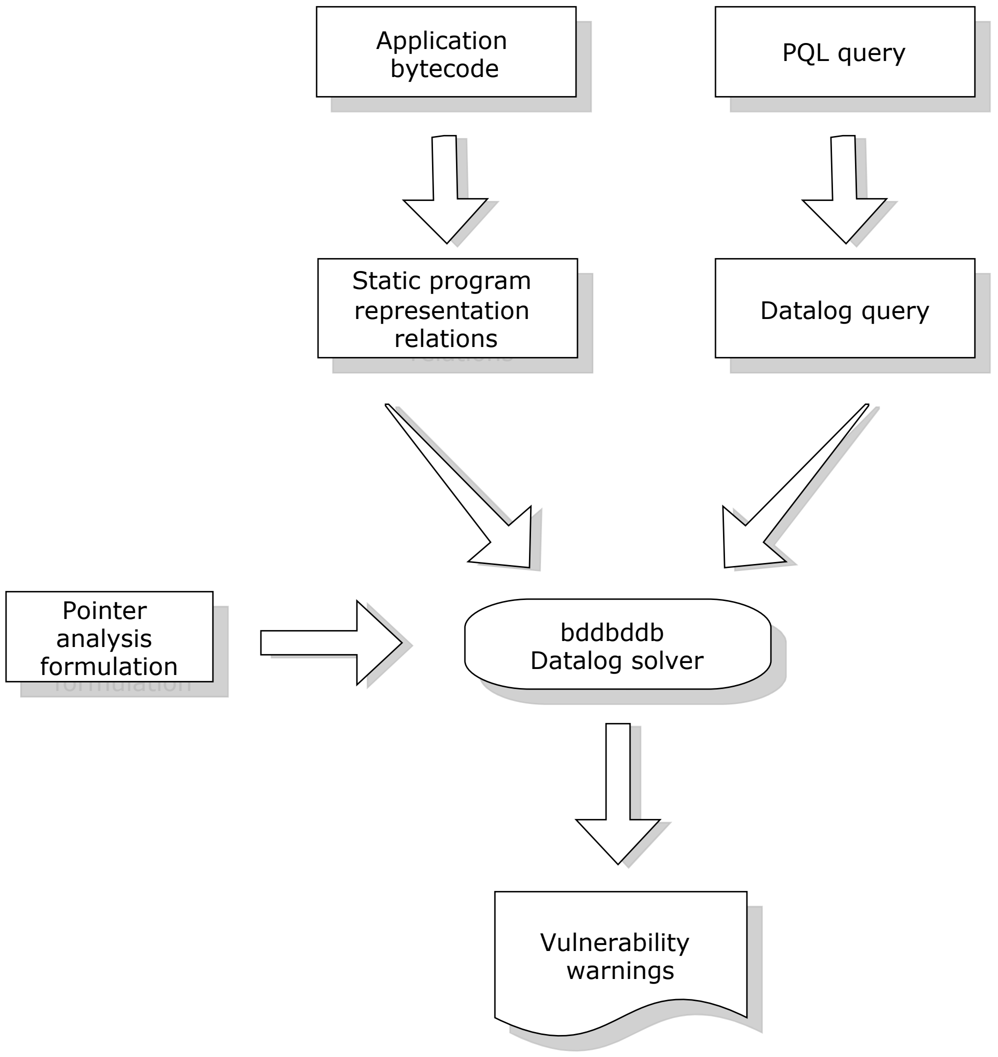


Figure 3.1: Architecture of the static analysis part of the Griffin project.

3.2 Motivating Example

Solutions of PQL queries can be found either using runtime or static analysis. At runtime, the solution to a PQL query can be found by observing program execution or by analyzing execution traces after the program finishes. Static analysis finds an *approximation* to the dynamic solution without executing the program. Static analysis captures all program execution to find all potential solutions satisfying the PQL query. However, the static solution may suffer from imprecision: some of the statically derived answers may be false positives.

The tainted object propagation problem described in Section 2.1.1 involves global data flow, which present a significant challenge for a static analysis tool. While local parameter passing in a program is relatively easy to track statically, an analysis that tracks the flow of data when it is deposited into fields of objects, arrays, or containers, such as hash maps, is considerably more complicated.

This is especially true in large enterprise software systems, where the use of multiple abstraction layers leads to data being passed around through multiple methods located across many files, directories, or libraries. A static analysis needs to be able to track this data passing precisely to avoid false positives. To further illustrate the difficulty of this problem, we begin with an example of code auditing for the purpose of security.

Example 3.1. For an example of the kind of work an auditor needs to perform to detect security vulnerabilities, consider a call to `sendRedirect` in `pebble`, a large J2EE online blogging application described in Chapter 6:

```
response.sendRedirect(blog.getUrl());
```

If the return result of `blog.getUrl`() is tainted, this call may lead to a HTTP splitting vulnerability. This is because the browser location the user is redirected to is under control of the attacker. Below is a sequence of steps that a code auditor has to perform by hand to determine if the return result of `blog.getUrl`() is tainted.

Step 1: Method `getUrl`() is implemented as follows:

```
public String getUrl() {
    return BlogManager.getInstance().getBaseUrl() + this.url;
}
```

Examining the implementations of method `getBaseUrl`() reveals that it returns the field `baseUrl`, which may be set by multiple calls to `setBaseUrl`(...).

Step 2: Similarly, `this.url` is set by multiple calls to `setUrl`(...).

Step 3: One of the many calls to method `setUrl`(...) looks as follows:

```
blog.setUrl(this.url + blog.getId() + "/");
```

Step 4: Method `getId`() is just a getter returning the field `id`.

Step 5: As it turns out, the setter method `setId`(...) is called with a parameter passed from the return result of `HttpRequest`.`getParameter`(`"id"`) through several levels of method invocation.

Step 6: As a result, the field `url` might be tainted, which in turns causes the return result of method `getUrl`() to be tainted as well.

Step 7: As a result, the return result of `getUrl`() may also be tainted.

However, arriving at this conclusion requires the examination of a great number of method calls spanning multiple files and directories. □

3.3 Static Analysis Overview

As mentioned above, a person auditing such an application by hand often quickly gets lost given the number of possibilities they need to consider. As a result, while simple errors will likely be detected, some of the more complex errors stemming from sources and sinks located "far apart" in the program may remain unnoticed. As our experiments summarized in Section 6.3.5 show, vulnerabilities that involve sources and sinks separated by many levels of abstraction are not uncommon. Similarly, automatic source code scanning tools that follow parameters or explore only a fixed number of paths though the program may miss potential vulnerabilities [28].

Our framework is based on a sound (i.e. conservative) analysis approach that finds *all* potential security violation that may be present in the program. The use of a sound analysis gives the application vendor or its user the assurance that an application for which no warnings is generated is in fact free of the type of security vulnerabilities the analysis is looking for. Our static analysis finds all potential violations matching a vulnerability specification given by its source, sink, derivation, and sanitization descriptors, as described in Section 2.1.1.

3.3.1 Static Program Representation

In this section we introduce the notation used to describe static analysis in the rest of this chapter.

Program as Relations

The input program is represented as a set of relations that capture the crucial aspects of program representation in the program, while omitting details that are not pertinent to the analysis of string data. For example, parameter passing will be captured, while integer manipulation will be omitted.

Conceptually, one can think of these relations as regular database relations. However, in order to save space, a specialized representation is used that uses binary decision diagrams (BDDs) to achieve high levels of compression for data with much commonality. Input relations that represent the program are stored in `bddbddb`, a BDD-based program database further described in Whaley et al. [205].

Relations in Datalog

The program database and the associated constraint resolution tool allows program analyses to be expressed in a succinct and natural fashion as a set of *rules* in Datalog, a logic programming language [192]. In the following, we say that predicate $R(x_1, \ldots, x_n)$ is true if tuple $(x_1, \ldots, x_n)$ is in relation R.

Elements of Datalog tuples are strictly typed, indicating which of several Datalog *domains* the element belongs to. Notation $R(x_1, \ldots, x_n) : D_1 \times \ldots \times D_n$ indicates that x_i is in Datalog domain D_i. The full list of Datalog domains used for program representation is shown in Figure 3.2. The list of input relations used to represent the program is shown in Figure 3.3.

Querying Program Properties Using Datalog

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

B	the domain of program bytecodes[1],
I	the domain of invocation sites,
V	the domain of program variables,
M	the domain of program methods,
H	the domain of heap objects named by their allocation site,
T	the domain of program types,
Z	the domain of integers, and
S	the domain of constant strings in the program.

Figure 3.2: Datalog domains used throughout this chapter.

rule

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

says that "$D(w, z)$ is true if predicates $A(w, x)$, $B(x, y)$, and $C(y, z)$ are all true." Datalog queries can be used to answer a variety of questions about static properties of the program, as show by the example below.

Example 3.2. Suppose we are interested in finding all methods that return `Cloneable` objects. We will define relation *ret-cloneable* : M as follows:

$$\begin{aligned} \textit{ret-cloneable}(m) \ :- \ & \textit{mret}(m, r), \textit{var-type}(r, t), \\ & \textit{subtype}(t, \texttt{"java.lang.Cloneable"}). \end{aligned}$$

Variable $m : M$ is the method returned from the query. Variable $r : V$ is the return value of method m, as captured by relation $\textit{mret}(m, r)$. Variable $t : T$ is the type of the return value r, as captured by $\textit{var-type}(r, t)$. Finally, type t must be a subtype of `java.lang.Cloneable`, as captured by $\textit{subtype}(t, \texttt{"java.lang.Cloneable"})$. □

Example 3.3. Suppose we are interested in finding all methods so that the first parameter's type and the return type are the same and is either a `String`, `StringBuffer`, or `StringBuilder`. Such methods are often either derivation or sanitization routines. To define relation *stringer* : M capturing this property, we will first define an auxiliary relation *str* : T as follows:

$$\begin{aligned}
\mathit{str}(t) &\;:-\; t = \texttt{"java.lang.String"}. \\
\mathit{str}(t) &\;:-\; t = \texttt{"java.lang.StringBuffer"}. \\
\mathit{str}(t) &\;:-\; t = \texttt{"java.lang.StringBuilder"}.
\end{aligned}$$

Relation *stringer* : M is defined as follows:

$$\mathit{stringer}(m) \;:-\; \mathit{mret}(m, r), \mathit{var\text{-}type}(r, t), \mathit{formal}(m, 1, p), \mathit{var\text{-}type}(p, t), \mathit{str}(t).$$

Relation *str* is used to factor out constraints on type variable $t : T$. □

3.3.2 Role of Pointer Information

Descriptors involved in a tainted object propagation problem in Section 2.1.1 refer to variables in the program code. To find security violations statically, it is necessary to know what runtime *objects* these descriptors may refer to, a general problem known as *pointer* or *points-to analysis*. 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 manipulation, as described in Example 1.

Example 3.4. In the code shown in Figure 3.4(a), string `userName` is tainted because it is returned from a source method `getParameter`. So is `buf1`, because it is derived from `userName` in the call to `append` on line 6. Finally, string `query` is derived from `buf2` and is subsequently passed to sink method `executeQuery`.

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 unsound optimistic tool will conclude that there no vulnerability is possible, potentially introducing a false negative. A similar situation arises in the process of auditing the code snippet in Figure 3.4(b). A vulnerability will be reported depending on whether `userName` and `query` may refer to the same object. Answering this question is far from obvious, as it depends on where the two lists

actual: $B \times I \times Z \times V$.	$actual(b, i, z, v)$ means that variable v is zth argument of the method call at i.
formal: $M \times Z \times V$.	$formal(m, z, v)$ means that variable v is zth parameter of the method m.
ret: $B \times I \times V$.	$ret(b, i, v)$, means that variable v is the return result of the method call at i.
thrown: $B \times I \times V$.	$thrown(b, m, v)$, means that variable v is an exception thrown by a method called at i.
mret: $M \times V$.	$mret(m, v)$, means that variable v is the return result of method m.
mthr: $M \times V$.	$mthr(m, v)$, means that variable v is an exception thrown by method m.
assign: $B \times V \times V$.	$assign(b, v_1, v_2)$ means that there is an implicit or explicit assignment statement $v_1 = v_2$ in the program.
load: $B \times V \times F \times V$.	$load(b, v_1, f, v_2)$ means that there is a load statement $v_2 = v_1.f$ in the program. Special symbol [ ] is used to encode array loads.
store: $B \times V \times F \times V$.	$store(b, v_1, f, v_2)$ means that there is a store statement $v_1.f = v_2$ in the program. Special symbol [ ] is used to encode array stores.
call: $B \times I \times M$.	$call(b, i, m)$ means that invocation site i may invoke method m.
var-type: $V \times T$.	$var\text{-}type(v, t)$ means that variable v has declared type t.
heap-type: $H \times T$.	$heap\text{-}type(h, t)$ means that heap allocation site h has declared type t.
subtype: $T \times T$.	$subtype(t_1, t_2)$ means that t_1 is a subtype of t_2.

Figure 3.3: Datalog relations used to represent the input program.

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

(a)

```
1.   String userName = req.getParameter("username");
2.
3.   LinkedList l1 = ...;
4.   LinkedList l2 = ...;
5.   ...
6.   buf1.add(userName);
7.   String query = (String) l2.getFirst();
8.   con.executeQuery(query);
```

(b)

Figure 3.4: Role of aliasing and points-to information.

declared on lines 3 and 4 are allocated and what elements they contain. □

Fortunately, the question of what *objects* a given program variable may refer to is precisely the question answered by pointer analysis. An unbounded number of objects may be allocated by the program at runtime, 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.

Example 3.5. In the program snipped in Figure 3.5, all 100 string objects allocated in the loop are approximated with the same allocation site on line 3. However, clashes introduced by object naming may or may not result in imprecision in the final results: in this case, *all* string objects allocated on line 3 will be either tainted on not, depending on whether `s1` is, so no imprecision is introduced. □

For every variable in the program, the analysis computes the sets of allocation sites that the variable may refer to. The basis of our approach is a context-sensitive

```
LinkedList l = new LinkedList();
for(int i = 0; i < 100; i++){
    String s2 = new String("string " + i) + s1;
    l.addLast(s);
}
```

Figure 3.5: Effect of object naming on precision of the results.

inclusion based points-to analysis that uses binary decision diagrams to make the computation and representation of results efficient [205]. Context sensitivity refers to the fact that the set of allocation sites a given variable may point to may in fact be different depending on the *invocation context* in which the variable is considered. The pointer analysis we use is generally flow-insensitive, which means that the same variable used in two different places in the program will have the same points-to set. Basic control flow, as applied to method locals, is handled in a flow sensitive way, as discussed in Section 3.4.6.

3.4 Pointer Analysis Precision Improvements

This section describes the role of various pointer analysis features towards improving the precision of static analysis results.

3.4.1 Role of Pointer Analysis Precision

Pointer analysis has been the subject of much compiler research over the last several decades [84]. Because accurately determining what heap objects a given program variable may point to during program execution is undecidable, sound analyses compute conservative approximations of the solution. Pointer analysis approaches typically trade scalability for precision, ranging from highly scalable but imprecise techniques [83, 182] to highly precise approaches that have not been shown to scale [116, 173].

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 [28, 80]. Such an approach, however, may miss important vulnerabilities, leading to false negatives.

The lack of precision provided by sound techniques 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. Below we describe analysis features that contribute to the precision of the results.

3.4.2 Role of Context Sensitivity

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. Context sensitivity also helps avoid the imprecision that is caused by mismatched calls and returns [168].

Imprecision of a Context-insensitive Analysis

Points-to analysis determines what heap objects variables in a method may point to [84]. However, the answer to this question can be heavily dependent on the invocation context the method is considered in. Unless different invocation contexts of method m are distinguished, formal arguments of the methods will point to all objects passed into the methods from different calls. Similarly, the return result of m will point to all objects that can possibly be returned from m. As a result, inputs from one call of m can flow to outputs of another call, a problem called *unrealizable paths* [113]. The effect of context sensitivity on analysis precision is illustrated by the example below.

Example 3.6. Consider the code snippet in Figure 3.6. The class `DataSource` acts as a wrapper for a URL string. The code creates two `DataSource` objects and calls the method `getUrl`() on both objects. A context-insensitive analysis would merge information for calls of `getUrl` on lines 16 and 17. The variable `this`, which is

```
 1.   class DataSource {
 2.       private String url;
 3.       public DataSource(String url) {
 4.           this.url = url;
 5.       }
 6.       String getUrl(){
 7.           return this.url;
 8.       }
 9.       ...
10.   }
11.   String passedUrl = request.getParameter("..."); // tainted
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 3.6: Example showing the importance of context sensitivity.

considered to be argument 0 of the call, points to objects allocated 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* variables `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. □

Cloning-based Context-Sensitive Pointer Analysis

While many points-to analysis approaches exist, until recently, we did not have a scalable context-sensitive conservative pointer analysis. The idea of using binary decision diagrams was explored by several researchers, including Zhu [213] and Berndl [21]. We rely on the context-sensitive, inclusion-based points-to analysis proposed by Whaley and Lam [205]. This analysis achieves scalability by using binary decision diagrams 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 the number of different invocation contexts is potentially infinite due to recursion, the pointer analysis we use approximates them with a finite set of context numbers for each methods. To do so, strongly-connected components of the call graph are "collapsed" into one node and paths reaching the method through the strongly connected components are numbered sequentially. 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.

A Datalog formulation of the pointer analysis due to Whaley and Lam is shown in Figure 3.7. Below we summarize the rules; interested readers are referred to Whaley et al. [205] for a more detailed explanation. Rule (1) is the base case of the points-to relation. Relations that come from objects being allocated, as captures by *points-to*$_0$ are added to relation *points-to*. Relation (2) captures a context-sensitive call relation, where the initial context numbers are copies from a call graph numbering relation *call-num*. Relations (4) — (7) are various assignment relations; (4) is simple interprocedural variable assignment, others are intraprocedural assignments, which also include data passing through exceptions. Finally, relations (8) — (10) are the pointer analysis relations that capture the interaction of assignments with *load* and *store* relations.

The Griffin static analysis framework is the first practical static approach for security to leverage a BDD-based pointer analysis for Java. The use of BDDs has allowed us to scale our framework to programs consisting of over 500,000 lines of code without sacrificing soundness. As our experimental evaluation if Section 6.3.3 shows, context sensitivity has a great impact on reducing the number of false positives.

3.4.3 Imprecision of Allocation Site Object Naming

Containers such as hash maps, vectors, lists, and others are a common source of imprecision for even a context-sensitive pointer analysis algorithm. The imprecision is due to objects being stored in data structures allocated *inside the container class definition*. These data structures often share the allocation site. As a result, pointer analysis cannot statically distinguish between objects stored in different containers. If a tainted value is deposited into one container of a given type, *all* values retrieved

$$\begin{array}{lll}
\textit{points-to}(\_, v, h) & :- & \textit{points-to}_0(v, h). \quad (1)\\
\textit{call}(vc_2, i, vc_1, m) & :- & \textit{call}_0(i, m), \textit{call-num}(vc_2, i, vc_1, m). \quad (2)\\
\textit{filter}(v, h) & :- & \textit{var-type}(v, tv), \textit{subtype}(tv, th), \textit{heap-type}(h, th). \quad (3)\\
\textit{assign}(\_, v_1, \_, v_2) & :- & \textit{assign}_0(v_1, v_2). \quad (4)\\
\textit{assign}(vc_1, v_1, vc_2, v_2) & :- & \textit{formal}(m, z, v_1), \textit{call}(vc_2, i, vc_1, m), \textit{actual}(i, z, v_2). \quad (5)\\
\textit{assign}(vc_2, v_2, vc_1, v_1) & :- & \textit{mret}(m, v_1), \textit{call}(vc_2, i, vc_1, m), \textit{return}(i, v_2). \quad (6)\\
\textit{assign}(vc_2, v_2, vc_1, v_1) & :- & \textit{mthr}(m, v_1), \textit{call}(vc_2, i, vc_1, m), \textit{thrown}(i, v_2). \quad (7)\\
\textit{points-to}(vc_1, v_1, h) & :- & \textit{assign}(vc_1, v_1, vc_2, v_2), \textit{points-to}(vc_2, v_2, h), \textit{filter}(v_1, h). \quad (8)\\
\textit{hpoints-to}(h_1, f, h_2) & :- & \textit{store}(v_1, f, v_2), \textit{points-to}(vc_1, v_1, h_1),\\
 & & \qquad \textit{points-to}(vc_1, v_2, h_2). \quad (9)\\
\textit{points-to}(vc_1, v_2, h_2) & :- & \textit{load}(v_1, f, v_2), \textit{points-to}(vc_1, v_1, h_1), \quad (10)\\
 & & \qquad \textit{hpoints-to}(h_1, f, h_2), \textit{filter}(v_2, h_2).
\end{array}$$

Figure 3.7: Context-sensitive pointer analysis rules of Whaley and Lam [205].

from *all* containers of the same type *anywhere else* in the program will be considered potentially tainted by a context-sensitive pointer analysis. The example below further illustrates the imprecision that comes from analyzing containers.

Example 3.7. An abbreviated version of the `Vector` class in shown in Figure 3.8. The constructor of the class allocates an internal array called `table` on line 4. In the client code, `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`.

A more formal description of how this conclusion is reached follows. Relations

```
 1.    public class Vector {
 2.      Object[] table = null;
 3.      public Vector(){
 4.          this.table = new Object[1024];
 5.       }
 6.
 7.       void add(Object value){
 8.           int i       = ...;
 9.           Object[] t1 = this.table;
10.           t1[i]        = value;
11.       }
12.
13.       Object getFirst(){
14            Object[] t2  = this.table;
15.           Object value = t2[0];
16.           return value;
17.       }
18.     }
19.     String s1 = "...";
20.     Vector v1 =  new Vector();
21.     v1.add(s1);
22.     Vector v2 =  new Vector();
23.     String s2 = v2.getFirst();
```

Figure 3.8: A sample `Vector` class definition and usage.

load(`this`, `table`, `t2`), *store*(`this`, `table`, `t1`), and *load*(`t2`, [ ], `value`) are obtained directly from the program code. Let h_n represent the allocation site of line n. We shall also represent the calling context by the line of code on which the call occurs.

1. We have *points-to*(_, `v1`, h_{19}) and *points-to*(_, `v2`, h_{21}), which are part of the initial points-to relation *points-to*$_0$ from pointer analysis rule (1).

2. Using the formal-actual propagation rule (5) in combination with rule (8) allows us to conclude *points-to*(c_{22}, `this`, h_{22}), where context c_{22} correspond to the call to method `getFirst`() on line 22 and `this` is a variable in method `getFirst`().

3. Relation *store*(`this`, `table`, `t1`) combined with *points-to*(c, `this`, h_{22}) leads to the heap points-to relation *hpoints-to*(h_{22}, `table`, h_4) according to rule (10).

4. A combination of rules (6) and (8) leads us to conclude that *points-to*(c_{21}, `value`, h_{19}) since *points-to*(_, `s1`, h_{19}) follows from rule (1).

```
public org.odmg.Transaction newTransaction() {
    try {
        return new Transaction( currentDatabase() );
    }
    catch (ODMGException ode) {
        throw new ODMGRuntimeException( ode.getMessage() );
    }
}
```

Figure 3.9: A sample factory function from the `hibernate` library, version 2.1.

5. Relations $\mathit{hpoints\text{-}to}(h_{22}, \texttt{table}, h_4)$ and $\mathit{load}(\texttt{this}, \texttt{table}, \texttt{t2})$ lead to $\mathit{points\text{-}to}(c_{23}, \texttt{t2}, h_4)$, according to rule (10).

6. Relation $\mathit{store}(\texttt{t2}, [\,], \texttt{value})$ combined with $\mathit{points\text{-}to}(c_{21}, \texttt{value}, h_{19})$ leads to the heap points-to relation $\mathit{hpoints\text{-}to}(h_{22}, \texttt{table}, h_4)$ according to rule (9).

7. Relations $\mathit{hpoints\text{-}to}(h_4, [\,], h_{19})$, $\mathit{points\text{-}to}(c_2, \texttt{table}, h_4)$, and $\mathit{load}(\texttt{t2}, [\,], \texttt{value})$ lead to $\mathit{points\text{-}to}(c_2, \texttt{value}, h_{19})$, according to rule (10).

8. Finally, a combination of rules (6) and (8) leads us to conclude that $\mathit{points\text{-}to}(c_2, \texttt{s2}, h_{19})$.

The final relation $\mathit{points\text{-}to}(c_2, \texttt{s2}, h_{19})$ implies that the string object allocated on line 19 may be the result result of a call to `getFirst`(). □

Containers are one of several programming constructs where the precision of a context-sensitive analysis is not enough. Another common programming idiom is *factory functions.*

Example 3.8. Consider factory method `newTransaction` extracted from the `hibernate` library shown in Figure 3.9. The default allocation site object naming scheme leads to imprecision in this case. All objects returned from method `newTransaction` will be associated with the allocation site within the method. This might produce a less precise answer compared to an equivalent piece of code in which allocations happen at the caller. This problem is sometimes referred to as allocation site burying [23]. □

```
public String getAddForeignKeyConstraintString(
    String constraintName, String[] foreignKey,
    String referencedTable, String[] primaryKey)
{
    return new StringBuffer(30)
    .append(" add constraint ")
    .append(" foreign key (")
    .append( StringHelper.join(StringHelper.COMMA_SPACE, foreignKey) )
    .append(") references ")
    .append(referencedTable)
    .append(" constraint ")
    .append(constraintName)
    .toString();
}
```

Figure 3.10: A sample string-manipulation routine from the `hibernate` library, version 2.1.

Imprecise object naming has especially severe consequences when it comes to string manipulation, as demonstrated by the example below.

Example 3.9. Consider a piece of code extracted from the `hibernate` library in Figure 3.10. Notice that the return result of method `getAddForeignKeyConstraintString` is tainted only if one of the parameters used in string concatenation is. However, there is only one `StringBuffer` object allocated within this method, leading to tainted and untainted values colliding.

Moreover, in this case method `StringBuffer.toString`() returns a newly allocated `String` object. Unless a better object naming scheme is used, as long as a single tainted object is passed into `StringBuffer.toString`(), *all* `String` objects returned from it will be tainted. Obviously, since our static analysis focuses on strings, this conclusion is grossly imprecise for most reasonably-sized programs. □

3.4.4 Improved Object Naming

Our approach to object naming involves judicious inlining of certain allocation sites. As the example in Figure 3.9 shows, inlining the factory routine achieves the same effect as parameterizing the allocation site with a calling context. Sometimes, one level of inlining is sufficient. Other times, several levels of inlining are necessary, as illustrated by the example in Figure 3.10. Inlining corresponds to parameterizing the allocation site by a list of call sites. While achieving the necessary precision in practice, the inlining-based approach suffers from the following shortcomings:

- **Need to know which allocation sites to inline.** Determining that is far from obvious. Moreover, if we decide to inline too little, this will result in imprecision. If we decide to inline too much, because many methods will be inlined into their calling context, the size of the program we need to analyze will expand, causing scalability problems.
- **Virtual calls with more than one target.** Call sites that *may* invoke a method of interest present a challenge. Inlining *all* methods that such a site may invoke often results in a significant expansion of the code under analysis.
- **Recursion.** There is no way to inline recursive methods while remaining conservative.

```
Input:  $M_0$
Output: $M$

$M \leftarrow M_0$;
$F \leftarrow \emptyset$;

do {
    $F_\Delta \leftarrow$ getFalsePositives() $\setminus F$;
    $A \leftarrow$ getAllocationSitesLeadingTo($F_\Delta$);

    foreach ($a \in A$) {
        $m \leftarrow$ getContainingMethod($a$);
        if ($\neg$ isRecursive($m$)) {
            $M \leftarrow M \cup \{m\}$;
        }
    }

    $F \leftarrow F \cup F_\Delta$;
} while ($F_\Delta \neq \emptyset$);
```

Figure 3.11: Incremental algorithm for determining methods to inline.

```
        ...
        String s = o.m();
         ...
        class C1 {
           ...
           String m(){
               return new String(...);
           }
        }
        class C2 {
           ...
           String m(){
               return new String(...);
           }
           ...
        }
```

Figure 3.12: Inlining a virtual site with multiple targets.

While there is no good solution for recursive methods, our approach to the other two challenges is described below.

A Semi-Automatic Technique for Determining Allocation Sites to Inline

Our algorithm for deciding what methods to inline is shown in Figure 3.11. The output of the algorithm is a set of methods M that are subject to inlining.

The initial set of methods to inline M_0 is determined experimentally and contains constructors of standard library container classes such as `HashMap`, `LinkedList`, etc. Moreover, since a method such as `HashMap.put`(...) may transitively call a method that allocates more elements of an array internal to the `HashMap`, calls to `HashMap.put`(...) and several other `HashMap` mutators need to be inlined as well. Furthermore, many standard library `String` and `StringBuffer` routines require inlining, as illustrated by Example 9. Set M_0 also contains several factory methods that are commonly used in J2EE applications.

The approach to selecting methods to inline is not fully automatic. In particular, method `getFalsePositives` requires human intervention to determine which of the reported warnings are in fact false positives. The amount of effort involved in

```
String s;
if (...) {
    s = o.m();          // call method C2::m only
} else {
    s = new String();   // inlined from line 7
}
```

Figure 3.13: Expanded call site to a virtual method with more than one target.

determined that is often non-trivial for some of the longer vulnerability traces.

Similarly, determining which allocation sites lead to the false positives in question, represented by method `getAllocationSitesLeadingTo` here is far from obvious. We have built tools that are able to automatically determine which call sites among the ones leading to a vulnerability trace have too many variables pointing to them. This appears to be a good heuristic for finding allocation sites to inline.

The construction of a fully automatic tool that finds candidates for selective inlining remains a subject of future work. However, our experience suggests that inlining certain well-chosen allocation sites is enough to achieve the necessary precision. What those sites are may vary depending on the property of interest as well as the program being analyzed.

Inlining Calls With More Than One Target

Traditional notion of inlining considers a call site invoking a single method. In object-oriented languages, virtual method dispatch violates this abstraction. Even the most precise call graph resolution algorithms in the worst case have call sites with more than one target method. Our approach is to only inline the callee methods of interest, while preserving the original call site. This is further illustrated by the example below:

Example 3.10. Consider the code snippet in Figure 3.12. Let us assume that object `o` on line 2 may refer to either class `C1` or class `C2` and we only want to inline method `C1.m`(). This will result in line 2 being replaced by the code shown in Figure 3.13. Since our analysis is oblivious to predicates, the exact value of the predicate used to choose between the inlined and the non-inlined version is immaterial. However, one

```
{
    ...
    // populates user information to return as a result
    Hashtable result = new Hashtable();
    result.put("nickname", user.getUserName());
    result.put("userid", user.getUserName());
    result.put("url", contextUrl+"/page/"+userid);
    result.put("email", "");
    result.put("lastname", lastname);
    result.put("firstname", firstname);

    return result;
    ...
}
```

Figure 3.14: Example from `roller` requiring precise map value handling.

way to think about it is as explicitly encoding the virtual dispatch logic. □

3.4.5 Improved Handling of Constant-Key Maps

A coding idiom common to large Java applications consists of having global map structures that use constant string keys. An example of this idiom extracted from a real-life application is shown below.

Example 3.11. This example is extracted from `roller`, a large open-source blogger written in Java. More information about `roller` may be found in Chapter 6. Shown in Figure 3.14 is a code excerpt that saves important information about the user in a `Hashtable`, which is subsequently passed around the program.

Individual values are referred to through constant string keys they are associated with. This idiom is often used in Java to create ad-hoc extensible `structs`. Clearly, being unable to distinguish between values associated with different keys will lead to less precise results when large maps with a variety of string data are used. □

Standard interprocedural propagation rules, rules (5) and (6), of pointer analysis shown in Figure 3.7 are augmented in order to handle maps with constant string keys. The new augmented rules shown in Figure 3.15 handle `HashMap`'s `put` and `get`

$$
\begin{array}{lll r}
\multicolumn{3}{l}{\#\ \textit{auxiliary predicates}} & \\
\textit{isGet}(m) & :- & m = \texttt{"HashMap.get(Object)"}. & (1) \\[1em]
\textit{constStr}(h) & :- & \textit{string2constant}(h, \_). & (2) \\[1em]
\textit{nonConstStr}(h) & :- & \neg \textit{constStr}(h). & (3) \\[1em]
\multicolumn{3}{l}{\#\ \textit{record calls to } \texttt{HashMap.put} \textit{ with a constant key}} & \\
\textit{put}(\textit{map}_h, \textit{key}_{str}, \textit{value}_h) & :- & & (4) \\
& & \textit{call}(c, i, \_, \texttt{"HashMap.put(Object,Object)"}), & \\
& & \textit{actual}(i, 1, \textit{key}), \textit{actual}(i, 2, \textit{value}), \textit{actual}(i, 0, \textit{map}), & \\
& & \textit{points-to}(c, \textit{key}, \textit{key}_h), \textit{points-to}(c, \textit{value}, \textit{value}_h), & \\
& & \textit{points-to}(c, \textit{map}, \textit{map}_h), & \\
& & \textit{string2constant}(\textit{key}_h, \textit{key}_{str}). & \\[1em]
\multicolumn{3}{l}{\#\ \textit{augmented return assignment rule}} & \\
\textit{assign}(vc_2, v_2, vc_1, v_1) & :- & \neg \textit{isGet}(m), & (5) \\
& & \textit{mret}(m, v_1), \textit{call}(vc_2, i, vc_1, m), \textit{return}(i, v_2). & \\[1em]
\multicolumn{3}{l}{\#\ \textit{retrieve from map with a constant key}} & \\
\textit{points-to}(vc_1, \textit{value}, \textit{value}_h) & :- & \textit{isGet}(m), \textit{string2constant}(\textit{key}_h, \textit{key}_{str}), & (6) \\
& & \textit{mret}(m, v_2), \textit{call}(vc_1, i, vc_2, m), \textit{return}(i, \textit{value}), & \\
& & \textit{actual}(i, 1, \textit{key}), \textit{actual}(i, 0, \textit{map}), \textit{filter}(\textit{value}, \textit{value}_h), & \\
& & \textit{points-to}(vc_2, \textit{key}, \textit{key}_h), \textit{points-to}(vc_2, \textit{map}, \textit{map}_h), & \\
& & \textit{put}(\textit{map}_h, \textit{key}_{str}, \textit{value}_h). & \\[1em]
\multicolumn{3}{l}{\#\ \textit{retrieve from a map with a non-constant key}} & \\
\textit{points-to}(vc_1, v_1, h) & :- & \textit{isGet}(m), \textit{nonConstStr}(\textit{key}_h), & (7) \\
& & \textit{mret}(m, v_2), \textit{call}(vc_1, i, vc_2, m), \textit{return}(i, v_1), & \\
& & \textit{actual}(i, 1, \textit{key}), \textit{points-to}(vc_2, \textit{key}, \textit{key}_h), & \\
& & \textit{points-to}(vc_2, v_2, h), \textit{filter}(v_1, h). &
\end{array}
$$

Figure 3.15: Rules for map sensitivity in pointer analysis.

methods in a special manner[2]. The following additional input relation is used to capture the notion of string equality:

string2constant: $H \times S$. *string2constant*(h, s) means that the string created at allocation site h is string constant s.

This additional relation is needed because the same string literal corresponds to different allocation sites when it appears in different parts of code.

Example 3.12. In the following piece of code

```
m.put("key", v1);   // h1
m.put("key", v2);   // h2
```

both lines will create a new allocation site for string `"key"`. The following two relations will be created for this code snippet:

$$\textit{string2constant}(h_1, \texttt{"key"}).$$

and

$$\textit{string2constant}(h_2, \texttt{"key"}).$$

and used by the pointer analysis rules in Figure 3.15. □

The pointer analysis rules for precise map handling are shown in Figure 3.15. Rules (1) — (3) are auxiliary relations used in the rest of the map-related rules. Relation *put* defined by rule (4) records when a value is associated with a constant string key. The standard pointer analysis interprocedural assignment rule, rule (6) in Figure 3.7 is augmented to *not* propagate values through `HashMap.get(...)` calls in rule (5). Rules (6) and (7) handle returning values associated with constant and non-constant keys, respectively.

3.4.6 Flow and Path Sensitivity

Local variables in a method are analyzed in a flow-sensitive manner before the pointer analysis is run; essentially, the same stack location is not confused as it is reused

[2]Other types implementing the `Map` interface can be treated in a manner similar to `HashMap`.

throughout the method. This allows the flow-insensitive interprocedural algorithm to take advantage of some local flow sensitivity, which is essential for accurately keeping track of updates to local variables [204].

An important characteristic of our static taint propagation is that it deals with string-like data. In Java, `String` objects are immutable. Moreover, `StringBuffer`s and `StringBuilder`s are monotonic, i.e. there is no way to *remove* data from a string data other than by calling method `StringBuffer.delete`. However, calls to this method have never been encountered in our benchmarks.

Therefore, once a `String` or a `StringBuffer` is deemed tainted by the analysis, it cannot be untainted with regular string operations unless a sanitizer is applied to it. As a result, a flow-insensitive analysis formulation where we talk about a static object approximation regarding of the program point is fully appropriate here.

It is important to point out that our pointer analysis as well as further analysis stages are oblivious to *predicates* found in the program (this is also sometimes referred to as *path sensitivity*). While false positives might result from not paying attention to predicates, we did not find this analysis feature useful in our set of benchmarks, as the results shown in Chapter 6 demonstrate. Of course, the value of predicate sensitivity varies greatly depending on the property of interest, so our practical observations should not be taken to mean that predicates have no value in static analysis for Java beyond the information flow properties we consider.

3.5 Finding Security Violations Statically

The presence of points-to information enables us to find security violations statically. Notice that the definition below closely mirrors the definition of a security violation given in Section 2.1.3.

Definition 3.5.1 A *static security violation* is a sequence of heap allocation sites $h_1, \dots, h_k$ such that

1. There exists a variable v_1 such that *points-to*$(\_, v_1, h_1)$, 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 $\textit{points-to}(\_, v_k, h_k)$, 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

$$\underset{1\leq i<k}{\forall} : \textit{points-to}(c_i, v_i, h_i) \wedge \textit{points-to}(c_i, v_{i+1}, h_{i+1}),$$

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 $\langle m, n_s, p_s, n_d, p_d\rangle$.

3.5.1 Static Checker and Optimizer

This section discusses how the tainted object propagation analysis is implemented in practice. Constraints of a specification as given by Definition 3.5.1 are 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, as described in Section 3.3.1. The points-to results used by the constraints are also readily available as Datalog relations after pointer analysis has been run. Points-to results serve as a link that connects heap object approximations and variables in the program; variables are in turn connected to method parameters, return values, etc.

Notice that 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 on the portion of the call graph that is statically analyzed; however, determining what classes 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 an analysis that handles common uses of reflection to increase the size of the analyzed call graph. The issue of reflection resolution is further addressed in Chapter 4.

3.5.2 Simple SQL Injection Query Translated

We start the discussion of how PQL vulnerability specifications are translated into Datalog by considering a simple PQL example.

Example 3.13. Figure 2.3 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 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 $\langle$`param`, `derived`$\rangle$ returned by the query. The **matches** clause of the query 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`

$$\begin{aligned}
&\mathit{simpleSQLInjection}(h^{\mathit{param}}, h^{\mathit{derived}}) \;:- && (1)\\
&\qquad \mathit{ret}(\_, i_1, v_1),\\
&\qquad \mathit{call}(\_, c_1, i_2, \texttt{"HttpServletRequest.getParameter"}),\\
&\qquad \mathit{points\text{-}to}(c_1, v_1, h^{\mathit{param}}),\\
&\\
&\qquad \mathit{actual}(\_, i_2, v_2, 0), \mathit{actual}(\_, i_2, v_3, 1), && (2)\\
&\qquad \mathit{call}(\_, c_2, i_2, \texttt{"StringBuffer.append"}),\\
&\qquad \mathit{points\text{-}to}(c_2, v_2, h^{\mathit{temp}}), \mathit{points\text{-}to}(c_2, v_3, h^{\mathit{param}}),\\
&\\
&\qquad \mathit{actual}(\_, i_3, v_4, 0), \mathit{ret}(\_, i_3, v_5), && (3)\\
&\qquad \mathit{call}(\_, c_3, i_3, \texttt{"StringBuffer.toString"}),\\
&\qquad \mathit{points\text{-}to}(c_3, v_4, h^{\mathit{temp}}), \mathit{points\text{-}to}(c_3, v_5, h^{\mathit{derived}}),\\
&\\
&\qquad \mathit{actual}(\_, i_4, v_6, 0), \mathit{actual}(\_, i_4, v_7, 1), && (4)\\
&\qquad \mathit{call}(\_, c_4, i_4, \texttt{"Connection.execute"}),\\
&\qquad \mathit{points\text{-}to}(c_4, v_6, h^{\mathit{con}}),\\
&\qquad \mathit{points\text{-}to}(c_4, v_7, h^{\mathit{derived}}) \;.
\end{aligned}$$

Figure 3.16: Simple SQL injection query translated into Datalog.

and `derived` matching the objects in `userName` and `query`. Query variable `temp` corresponds to the temporary `StringBuffer` created by the Java compiler for the string concatenation operation in Example 1.

The result of translating `simpleSQLInjection` into Datalog is shown in Figure 3.16. To make the correspondence between the original PQL query and the generated Datalog query more apparent, object x in PQL is approximated by allocated site h^x in Datalog. Parts (1) — (4) of the Datalog translation correspond directly to the four lines of the **matches** clause of the original PQL query. Notice that the resulting Datalog demands that the contexts in which various *points-to* relations corresponding to single PQL event match. As can be seen from the example above, the resulting Datalog is quite involved even for a relatively simple query and is therefore not a very good specification language for describing vulnerabilities. □

3.5.3 Systematic Translation from PQL into Datalog

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

To make our discussion more systematic, the reader is referred to a BNF grammar for PQL queries shown in Figure 2.4. The topmost level non-terminal is *queries*, which stands for a list of PQL queries. In the rest of the discussion, we describe how each individual query is translated from PQL into Datalog.

Normalizing the PQL Query

In the beginning of the translation process, we *normalize* the input PQL queries so that the **matches** part of each query is an alternation of sequence statements. In other words, the topmost level statement of the **matches** clause is an *altStmt* each of whose clauses is a *seqStmt* expression. Moreover, *altStmt*s mentioned in Figure 2.4 are used *only* at the top level.

It should be pointed out that the other parts of a PQL query captured by the grammar in Figure 2.4 are ignored in the translation. Indeed, **replaces** and **executes** clauses that insert runtime events to execute on a query match only pertain to the runtime system, as explained in Section 5.5. The **returns** clause is used to determine which variables need to be included in the resulting Datalog query and which can be safely dropped. Finally, Datalog does not require a special variable declaration, so the **uses** clause can be dropped.

The normalization process is performed by introducing new auxiliary subqueries as necessary, which may increase the size of the input, but greatly simplifies the translation. Any event affected by a **replaces** clause is treated by this process as being a possible final event in the query. This is equivalent to appending an alternation of all such statements to the end of the **matches** clause before normalization. Since PQL variables refer to objects and our static analysis uses allocation sites as an approximation of object identity, for every reference to object h in PQL, h is replaced with a Datalog variable v_h and require that v_h point to h: $\textit{points-to}(c, v_h, h)$. Literals

PQL grammar transition	**Datalog translation**
$\textit{primStmt} \longrightarrow \textit{fieldAccess} = \textit{id}$	$\textit{store}(\_, v_1, \textit{fieldAccess.id}_2, v_2)$, $\textit{points-to}(c, v_1, \textit{fieldAccess.id}_1)$, $\textit{points-to}(c, v_2, \textit{id})$
$\textit{primStmt} \longrightarrow \textit{id} = \textit{fieldAccess}$	$\textit{load}(\_, v_1, \textit{fieldAccess.id}_2, v_2)$, $\textit{points-to}(c, v_2, \textit{fieldAccess.id}_1)$, $\textit{points-to}(c, v_1, \textit{id})$
$\textit{primStmt} \longrightarrow \textit{id}[\,] = \textit{id}$	$\textit{store}(\_, v_1, [\,], v_2)$, $\textit{points-to}(c, v_1, \textit{id}_1)$, $\textit{points-to}(c, v_2, \textit{id}_2)$
$\textit{primStmt} \longrightarrow \textit{id} = \textit{id}[\,]$	$\textit{load}(\_, v_1, [\,], v_2)$, $\textit{points-to}(c, v_2, \textit{id}_1)$, $\textit{points-to}(c, v_1, \textit{id}_2)$
$\textit{primStmt} \longrightarrow \textit{id} = \textit{methodName}\ (\ \textit{idList}\ )$	$\textit{ret}(i, \textit{id}), \textit{call}(c, i, \_, \textit{methodName})$, $\textit{actual}(i, 1, v_1), \textit{points-to}(c, v_1, \textit{idList.id}_1)$, $\ldots$ $\textit{actual}(i, n, v_n), \textit{points-to}(c, v_n, \textit{idList.id}_n)$
$\textit{primStmt} \longrightarrow \textit{id} = \texttt{new}\ \ \textit{typeName}\ (\ \textit{idList}\ )$	$\textit{ret}(i, \textit{id}), \textit{call}(c, i, \_, \textit{typeName}.\texttt{<init>})$, $\textit{actual}(i, 1, v_1), \textit{points-to}(c, v_1, \textit{idList.id}_1)$, $\ldots$ $\textit{actual}(i, n, v_n), \textit{points-to}(c, v_n, \textit{idList.id}_n)$

Figure 3.17: Translation of primitive statements *primStmt* from PQL (left) into Datalog (right) for static analysis and optimization.

and wildcards are translated from PQL into Datalog without change.

Handling PQL Constructs

We summarize the handling of individual PQL constructs below:

- **Primitive statements**. Each primitive statement in the query is translated into one or more Datalog predicates. A syntax-directed translation of PQL queries into Datalog is shown in Figure 3.17. The left side of the table lists the PQL grammar rule and the right hand side shows its Datalog translation.

  Indices are used to disambiguate between different identifiers id_1 and id_2 used in PQL rules. Nonterminal *fieldAccess* expands into $id_1.id_2$ and *fieldAccess*.id_i notation is used to refer to each used identifier. Method invocation is the only PQL primitive that requires multiple Datalog predicates, as shown in Example 13. A special field [ ] is used to denote array access. Notation *typeName*.`<init>` is used to denote the constructor of type *typeName*.

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

- **Sequencing**. Because the static analysis is flow-insensitive, we do not track sequencing directly, and instead merely demand that all events in the sequence occur at some point. This can be done by simply replacing the sequence operator ";" with the Datalog conjunction operator ",". If a path represented by the sequence is shorter than the maximum-length path, the remaining bytecode parameters are forced to be equal to the special value ϵ, which stands for "no bytecode".

- **Independent blocks**. Similarly, because the static analysis is flow-insensitive, the translation of a comma statement can simply treat all its clauses as part of its containing sequence.

- **Exclusion**. Since our static analysis is flow-insensitive, no guarantees about event ordering can be made. This implies that we cannot deduce that an excluded event (denoted with a $\sim$) occurs between two points in a sequence; as a result, all excluded events are ignored. For more details, see a discussion of sanitizers in Section 3.5.5.

- **Unification**. Unification of PQL variables is translated into equality of heap allocation sites.
- **Subqueries**. Invocations of PQL subqueries are represented by referring to the equivalent Datalog relation. The program points and any variables that are not parameters in the PQL subquery are matched to wildcards and projected away.

3.5.4 Extracting the Relevant Bytecodes

The Datalog provided in the previous subsection finds all program points and heap variables for each subquery individually. The final result, however, requires extracting only those program points that actually participate in a top-level match.

This is a two-step process. In the first step, we determine which subquery invocations contribute to the final result; in the second we project the relevant subqueries onto the bytecode domain B to get a set of bytecodes that require to be instrumented.

1. **Finding relevant subquery values.** Relevant subquery values are determined inductively as follows:

   - any parameter of the query `main` is relevant, and
   - any subquery value that appears as a clause on the right hand side in a relevant relation is relevant.

   This can be translated directly to Datalog, with one rule for each subquery invocation and an additional rule to express that all parameters of query `main` are relevant.

2. **Extracting relevant program locations.** Gathering the relevant program locations is straightforward once the previous step is performed. Any program location that occurs in a relevant solution to any query is relevant. For the special case of the `main` query, we need not check for relevance because all solutions to the main query are relevant.

$$
\begin{aligned}
& \mathit{main}(b_1, b_2, b_3, b_4, h^{param}, h^{derived}) \;:- && (1)\\
&\qquad \mathit{ret}(b_1, i_1, v_1),\\
&\qquad \mathit{call}(b_1, c_1, i_2, \texttt{"HttpServletRequest.getParameter"}),\\
&\qquad \mathit{points\text{-}to}(c_1, v_1, h^{param}),\\
&\\
&\qquad \mathit{actual}(b_2, i_2, v_2, 0), \mathit{actual}(b_2, i_2, v_3, 1),\\
&\qquad \mathit{call}(b_2, c_2, i_2, \texttt{"StringBuffer.append"}),\\
&\qquad \mathit{points\text{-}to}(c_2, v_2, h^{temp}),\\
&\qquad \mathit{points\text{-}to}(c_2, v_3, h^{param}),\\
&\\
&\qquad \mathit{actual}(b_3, i_3, v_4, 0), \mathit{ret}(b_3, i_3, v_5),\\
&\qquad \mathit{call}(b_3, c_3, i_3, \texttt{"StringBuffer.toString"}),\\
&\qquad \mathit{points\text{-}to}(c_3, v_4, h^{temp}),\\
&\qquad \mathit{points\text{-}to}(c_3, v_5, h^{derived}),\\
&\\
&\qquad \mathit{actual}(b_4, i_4, v_6, 0), \mathit{actual}(b_4, i_4, v_7, 1),\\
&\qquad \mathit{call}(b_4, c_4, i_4, \texttt{"Connection.execute"}),\\
&\qquad \mathit{points\text{-}to}(c_4, v_6, h^{con}),\\
&\qquad \mathit{points\text{-}to}(c_4, v_7, h^{derived})\;.
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{mainRelevant}(h^{param}, h^{derived}) \;:- && (2)\\
&\qquad \mathit{main}(\_, \_, \_, \_, h^{param}, h^{derived}).\\
& \mathit{relevant}(b_1) \;:-\; \mathit{main}(b_1, \_, \_, \_, \_, \_,). && (3)\\
& \mathit{relevant}(b_2) \;:-\; \mathit{main}(\_, b_2, \_, \_, \_, \_,). && (4)\\
& \mathit{relevant}(b_3) \;:-\; \mathit{main}(\_, \_, b_3, \_, \_, \_,). && (5)\\
& \mathit{relevant}(b_4) \;:-\; \mathit{main}(\_, \_, \_, b_4, \_, \_,). && (6)
\end{aligned}
$$

Figure 3.18: Simple SQL injection query translated to extract the relevant bytecode information.

Example 3.14. Figure 3.18 shows the Datalog translation of Example 1 that is used to extract the relevant bytecodes. (For consistency, we renamed the main query to `main` instead of `simpleSQLInjection`.) Statement (1) shows the main query itself

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

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

Figure 3.19: Recursive query for tracking data from sockets.

translated with bytecode information encoded as $b_1, ..., b_4$ preserved. Statement (2) shows a projection of the main query to preserve the bytecode information for each component of the query in statement (1). Since there is no non-main subquery, this subquery does not strictly have to be introduce in this case. Finally, query *relevant* collects all of these bytecodes in statements (3) — (6). □

Example 3.15. As this example illustrates, the translation process is more complex for recursive queries. Recursion in PQL is useful for matching against the common idiom of *wrappers* in Java. Java exposes higher-level I/O functions by providing wrappers over base input streams. These wrappers are subclasses of the top-level interfaces `Reader` (for character streams) and `InputStream` (for byte streams).

For example, to read Java Objects from some socket `s`, one might first wrap the stream with a `BufferedInputStream` to cache incoming data, then with an `ObjectInputStream` to parse the objects from the stream:

$$
\begin{aligned}
&\mathit{main}(b_1, b_2, b_3, m, h_s, h_v, h_x, h_y) && :- && (1)\\
& && \mathit{call}(c_0, b_1, \_, \texttt{"getInputStream"}), &&\\
& && \mathit{actual}(b_1, 0, v_{s0}), \mathit{ret}(b_1, v_x), &&\\
& && \mathit{points\text{-}to}(c_0, v_x, h_x), \mathit{points\text{-}to}(c_0, v_{s0}, h_s), &&\\
& && \mathit{derivedStream}(\_, h_x, h_y, \_), &&\\
& && \mathit{call}(c_1, b_{2,}, \texttt{"readObject"}), &&\\
& && \mathit{actual}(b_2, 0, v_{y1}), \mathit{ret}(b_2, v_{v1}), &&\\
& && \mathit{points\text{-}to}(c_1, v_{v1}, h_v), \mathit{points\text{-}to}(c_1, v_{y1}, h_y), &&\\
& && \mathit{call}(c_2, b_3, \_, m), \mathit{actual}(b_3, 0, v_{v2}), &&\\
& && \mathit{points\text{-}to}(c_2, v_{v2}, h_v). &&\\
&\mathit{derived}(b, h_x, h_d, \_) && :- && (2)\\
& && h_x = h_d, b = \epsilon. &&\\
&\mathit{derived}(b, h_x, h_d, h_t) && :- && (3)\\
& && \mathit{call}(c, b, \_, \texttt{"InputStream.<init>"}), &&\\
& && \mathit{actual}(b, 1, v_x), \mathit{ret}(b, v_t), &&\\
& && \mathit{points\text{-}to}(c, v_x, h_x), \mathit{points\text{-}to}(c, v_t, h_t), &&\\
& && \mathit{derived}(\_, h_t, h_d, \_). &&
\end{aligned}
$$

Figure 3.20: Datalog translation of the PQL query in Figure 3.19.

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

In general, there can be arbitrary levels of wrapping. To capture this, we need to use a recursive pattern, as shown in Figure 3.19. The base case in *derivedStream* subquery declares that any stream can be considered derived from itself; the other captures a single wrapper and then re-invokes *derivedStream* recursively.

The query shown in Figure 3.19 finds all methods invoked on objects read from a network socket. The query first finds all the streams derived from the input stream of a socket, then all objects read from any of the derived streams. It then matches against any method, represented by the method parameter `m`, invoked upon the objects read.

Figure 3.21 uses the *relevant* relation to express this. The first rule says that

$$
\begin{array}{lll}
\mathit{mainRelevant}(m, h_s, h_v, h_x, h_y) & :- & \qquad (1)\\
 & & \mathit{main}(\_, \_, \_, m, h_s, h_v, h_x, h_y).\\
\mathit{derivedRelevant}(h_x, h_y, h_3) & :- & \qquad (2)\\
 & & \mathit{mainRelevant}(\_, \_, \_, h_x, h_y),\\
 & & \mathit{derived}(\_, h_x, h_y, h_3).\\
\mathit{derivedRelevant}(h_t, h_d, h_3) & :- & \qquad (3)\\
 & & \mathit{derivedRelevant}(\_, h_d, h_t),\\
 & & \mathit{derived}(\_, h_t, h_d, h_3).\\
\mathit{relevant}(b) & :- & \qquad (4)\\
 & & \mathit{derived}(b, h_x, h_d, h_t),\\
 & & \mathit{derivedRelevant}(h_x, h_d, h_t).\\
\mathit{relevant}(b) & :- & \mathit{main}(\_, \_, b, \_, \_, \_, \_, \_). \qquad (5)\\
\mathit{relevant}(b) & :- & \mathit{main}(\_, b, \_, \_, \_, \_, \_, \_). \qquad (6)\\
\mathit{relevant}(b) & :- & \mathit{main}(b, \_, \_, \_, \_, \_, \_, \_). \qquad (7)
\end{array}
$$

Figure 3.21: Extracting relevant bytecodes for the PQL query in Figure 3.19.

all non-bytecode parameters of *main* are relevant. Rules (2) and (3) in Figure 3.21 correspond to rules (2) and (3) in Figure 3.20. Recursive definition of *derivedRelevant* in rule (3) corresponds to recursive definition of *derived* in Figure 3.20. Finally, rules (4) — (7) extract the bytecodes that matter for the top-level query *main*. □

3.5.5 Dealing with Sanitizers

While detecting if a certain sanitizer has been applied to an object at runtime is quite easy, the fact that we only have *may* points-to information presents a problem.

Example 3.16. Consider the code snippet in Figure 3.22. Variable `str` will point to both the return result of `sanitize` as well as the original, tainted value `username`. According to our static security violation formulation in Section 3.5, we *will* report this case. However, our analysis will also happily report the piece of code shown in Figure 3.23 as a violation as well. However, this is a false positive because the assignment on line 3 overrides the old value of `str`. This false positive will result

```
String username = req.getParameter("username");
String s = null;

if (...) {
        str = sanitize(username);
} else {
        str = username;
}
```

Figure 3.22: Simple conditional sanitization example.

because our points-to information is flow-insensitive. However, we can still use points-to information to detect that `str` points to a string pointed to by the return result of method `sanitize`. □

To preserve soundness, our approach is to still report cases like this, but as lower-level warnings, as explained below:

Definition 3.5.2 A violation trace $h_1, ..., h_n$ is *low-level* if there exists a variable v such that v corresponds to applying p_d to argument n_d for a sanitization descriptor $\langle m, n_d, p_d \rangle$ and $\exists\ i : 0 \leq i \leq n$ such that *points-to*$(\_, v, h_i)$ holds.

Since most sanitization is performed by creating a new fresh sanitized object, omitting a sanitizer stops the propagation of taint, giving us the correct result. There are only a few low-level warnings in all our experiments.

3.6 Static Analysis Soundness

Our analysis finds all vulnerabilities in statically analyzed code that are captured by the specification.

3.6.1 Static Analysis Limitations

Failing to obtain a conservative call graph or having an incomplete specification compromises the static soundness of the analysis results, as discussed below.

```
String username = req.getParameter("username");
String str = username;
str = sanitize(username);
con.executeQuery("SELECT ... " + str);
```

Figure 3.23: False positive in our formulation.

Static Analysis Coverage

The static analysis approach needs to be able to analyze *all* code that might be executed at runtime. Otherwise, vulnerabilities might be missed. However, finding all the relevant code is far from obvious. We postpone the discussion of the analysis coverage issue until Section 3.7 and Chapter 4.

Specification Completeness

The analysis results are only complete if the specification is. In order for all vulnerabilities to be detected, the sets of sources, sinks, derivation, and sanitization descriptors must be complete. For instance, if derivation descriptors are missing, some portions of the taint propagation graph will be unexplored and some potential vulnerabilities may go unreported.

Our current approach delegates the task of creating complete lists of descriptors — or the PQL specification — to the user. While this might seem like a difficult task, our approach here is not very different from what is done by other work in the space of software error detection [9, 80, 194]. As described in Chapter 8.2, aiding the user with the task of specification creation is an important area of future research.

3.6.2 Troublesome Program Constructs

Below we discuss analysis issues and programming constructs that compromise the soundness of our approach. Our static analysis techniques are sound for programs devoid of such constructs.

```
String s = ...;
char[] chars = s.getChars();
for(int i = 0; i < chars.length; i++){
    if(chars[i] != '/') buf.append(chars[i]);
}
s = buf.toString();
// proceed to use string s
```

Figure 3.24: Inline character-level string manipulation

Inline Character-level String Manipulation

While this is much less common than in C, which represents strings as character arrays, Java programs still occasionally operate on strings at the level of individual characters. As discussed above, our analysis works at the level of *objects*, so the flow of taint through characters may be lost as a result.

Most character-level operations are encapsulated into methods. However, as the code in Figure 3.24 shows, it is possible to have string manipulation statements inline. Our framework, as described in Chapter 2, does not capture inline string manipulation, which we believe to be rare. A conservative approach would be to report all places where character-level manipulation may occur to be reviewed by the user.

Native Methods

Native methods may have side-effects that affect objects located on the Java heap. Since `native` methods are not written in Java and are hard to analyze, we do not support native methods possibility in our static analysis system. Moreover, similarly to the static analysis technique, the runtime approach also does not track data that is passed into or returned from native code.

Recent work has considered dangers involved in native function calls [60, 61, 175, 212]. However, we do not believe that string-manipulating `native` routines to be common. The majority of JDK's `String` and `StringBuffer` code is written in Java, with only a few `native` methods. One possible exception where the handling of `native` code *may* matter is method `System.arraycopy` that efficiently copies data between arrays.

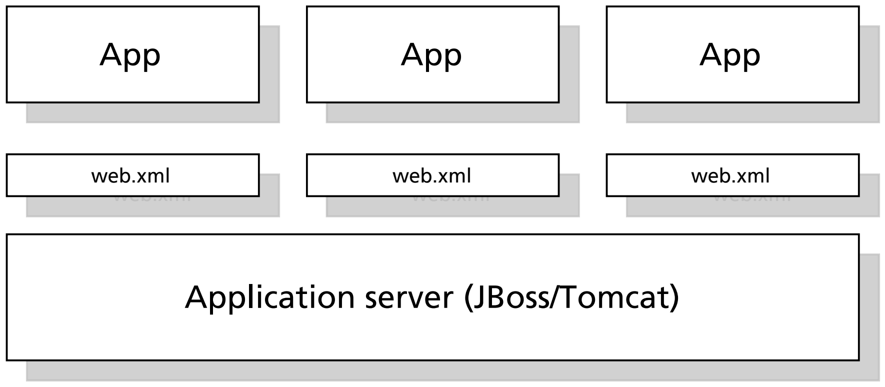


Figure 3.25: Typical application server architecture.

Data Flow Through Exceptions

For efficiency reasons, we currently disregard the data flow that is caused by exception passing in Java. Flow through exceptional paths is supported, as reflected in rule (7) of the pointer analysis in Figure 3.7. While it would not be too difficult to add, exceptional flow paths are mostly interesting for information leak vulnerabilities, which are outside the scope of our experiments.

3.7 Static Analysis Coverage

In this section we describe features of the analysis that allow us to increase the amount of code that is analyzed statically. It is helpful to start by contrasting whole-program analysis of a stand-alone Java program with that of a Web-based application.

3.7.1 Challenges of Call Graph Construction in Web Applications

Since Web applications are designed to be deployed on an application server such as Apache Tomcat or JBoss, static analysis of Web application code presents unique challenges not present in regular command line or GUI-based applications. Since it is generally not known what methods a given program consists of, whole-program static analysis of Java typically works by constructing a static call graph. This is usually done in two steps through a process called *call graph discovery* [205]:

1. A set of *root methods*, which are entry points into the call graph, such as `main` is identified by the user.

2. Call graph edges are added until a fixed point is reached when no more methods can be added to the call graph.

However, when the call graph discovery approach is used naïvely, the analysis is very hard to scale because too much code is included in the call graph. Shown in Figure 3.25 is a typical application server architecture. The size of a large Web-based Java application can be as much as 0.5 million lines of code. According to `Koders.com` statistics, the size of recent versions of JBoss, one of the more popular application servers, is over 1.5 million lines [106]. Combined together, this yields a program over 2 million lines of code in size. Performing a global and precise static analysis on such a large code base is a formidable task. We describe our approach to finding the call graph roots in the rest of this section and leave the discussion of challenges introduced into the process of call graph discovery when reflection is used until Chapter 4.

3.7.2 Finding Call Graph Roots

To keep our approach scalable, instead we choose to analyze the Web applications in a stand-alone 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 [190] or using mock objects for testing [137].

Finding all root methods in Web applications presents a challenge. J2EE-based applications are designed to run within a J2EE application server with the server invoking different components of the application. 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*

```
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 3.26: Invocation stub program generated for `blueblog`.

stub, a small driver program that invokes each servlet and action in the application in turn. The following components of the Web applications are handled within the generated stub:

- J2EE servlets,
- J2EE listeners,
- J2EE tags and taglibs,
- Apache Struts actions and forms.

While servlets are relatively easy to process, Java Server Pages (JSPs) present a particular challenge. JSPs are regular HTML with pieces of Java code embedded in them. The presence of Java code opens JSPs to security attacks. JSPs are typically distributed as source code and are only compiled down to Java bytecode at the time of application deployment. However, setting up a proper static compilation environment is often difficult. Another approach consists of deploying the application and then copying the compiled JSP bytecode from the temporary application server location. Our technique currently does not address Java beans, which would require an approach which is similar to that we use for reflection, as described in Chapter 4.

3.7.3 Using a Mock Library

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 stand-alone program. Passing in `null` parameters is not a viable possibility, as that would make all the relevant points-to sets empty. 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 [137]. While this approach allows us to scale to large applications, we may miss some vulnerabilities contained in application server sources, which are not analyzed. Moreover, the user of the tool has no way of estimating the number of vulnerabilities that may still be missing.

To generate an invocation stub, *Web application descriptors* usually 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* typically described by a `struts-config.xml` file [31].

Example 3.17. An example of a stub generated for a smaller Web-based application called `blueblog` is shown in Figure 3.26. Method `processServlets` constructs a mock `MyHttpServletRequest` and `MyHttpServletResponse` and passes them to method `service` of a newly constructed `BBServlet`. Typically, stubs are quite a bit larger, though, since many real-life J2EE applications contain dozens of servlets. □

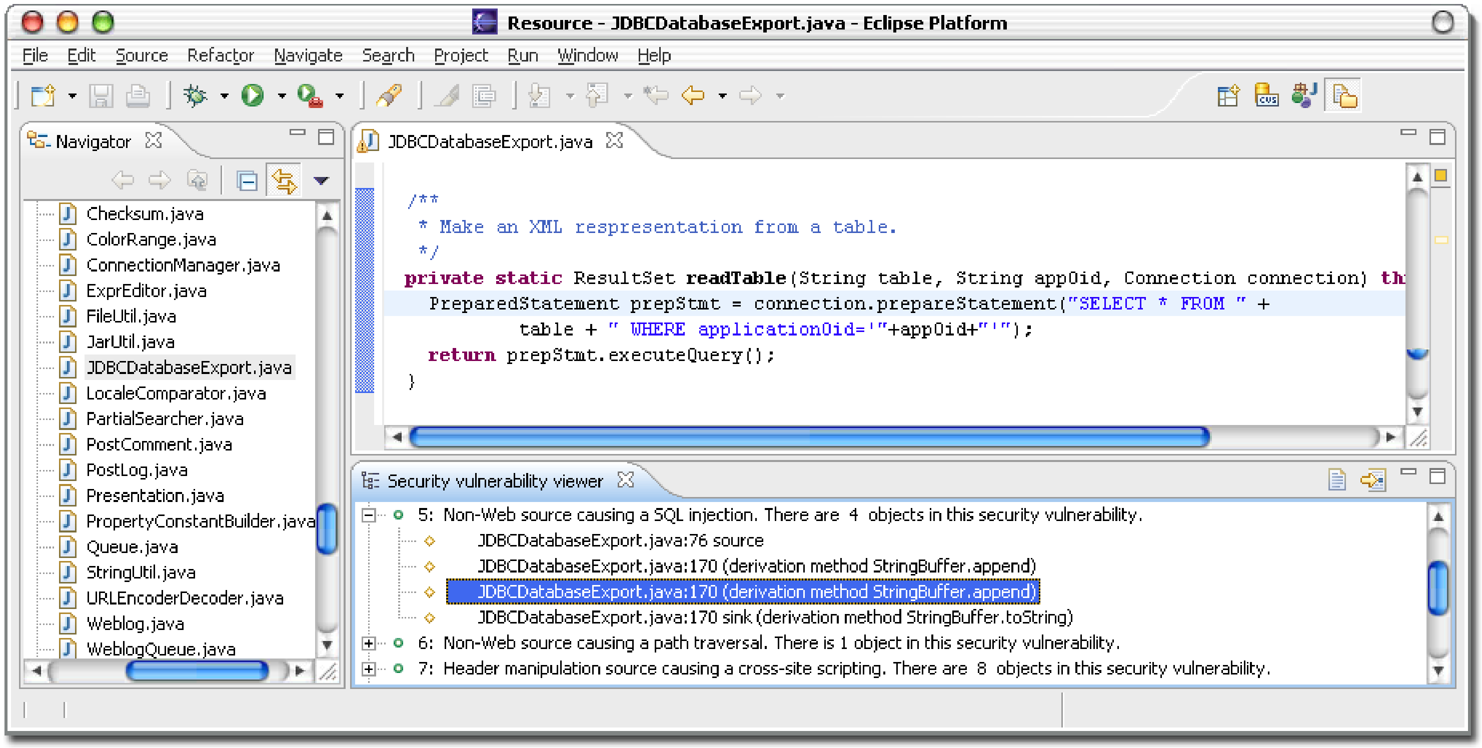
3.7.4 Limitations of This Approach

While the mock library approach gives us the scalability we want, it replaces the actual application server with a stylized harness, which might not have all of server's complexity. As a result, the following types of vulnerabilities might be missed with the harness-based approach:

- *Vulnerabilities inside application server code as opposed to application code will not be detected.* As a result, our approach should be seen as a technique for finding vulnerabilities *within* application code.

- *Vulnerabilities that only exhibit themselves when the application is deployed within a particular application server environment.* Our harness-based approach is a simplification of the complexity of a real application server. As a result of this specification and insufficiently accurate environment modeling, vulnerabilities might be missed.

While we do not believe these to be serious practical limitations of our approach, we mention them here for completeness.



```
/**
 * Make an XML respresentation from a table.
 */
private static ResultSet readTable(String table, String appOid, Connection connection) th
  PreparedStatement prepStmt = connection.prepareStatement("SELECT * FROM " +
             table + " WHERE applicationOid='"+appOid+"'");
  return prepStmt.executeQuery();
}
```

```
5:  Non-Web source causing a SQL injection. There are 4 objects in this security vulnerability.
        JDBCDatabaseExport.java:76 source
        JDBCDatabaseExport.java:170 (derivation method StringBuffer.append)
        JDBCDatabaseExport.java:170 (derivation method StringBuffer.append)
        JDBCDatabaseExport.java:170 sink (derivation method StringBuffer.toString)
6:  Non-Web source causing a path traversal. There is 1 object in this security vulnerability.
7:  Header manipulation source causing a cross-site scripting. There are 8 objects in this security vulnerability.
```

Figure 3.27: Eclipse plugin for reviewing analysis results.

3.8 User Interface to Analysis Results

To help the user with the task of examining violation reports, the Griffin project provides an intuitive GUI. 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 it involves. 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 3.18. An example of using the Eclipse GUI is shown in Figure 3.27. 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.

Each line of the expanded vulnerability display at the bottom of the screen represents a static object approximation within a static vulnerability trace. 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. Types of vulnerability as well as trace lengths are shown to simplify vulnerability classification. □

It is necessary to point out that a statically computed vulnerability trace consists of a list of *static object approximations*, which in our case are allocation sites. While line number information for each allocation site is generally available (library code can be compiled without line number information), the user often does not know what *variables* correspond to a given object.

Even if an object-to-variable mapping is extracted from the results of pointer analysis, it is often a challenge to determine what sequence of calls makes an object is one file derived from an object in another file or directory.

3.9 Chapter Summary

In this chapter we presented the static analysis portion of the Griffin project. Static analysis is an especially powerful technique when it comes to finding security errors because it can explore all possible execution paths without requiring an input. In this chapter we showed how our static technique relies on a points-to analysis. We introduced a range of pointer analysis precision improvements that result in a reduction of the number of false positives. This chapter also describes a scalable approach to analyzing Web applications independently of the application server.

Chapter 4

Analysis of Reflection

This chapter describes an approach to call graph construction for Java programs in the presence of reflection.

4.1 Introduction

Whole-program static analysis requires knowing the targets of function or method calls. The task of computing a program's call graph is complicated for a language like Java because of virtual method invocations and reflection. Past research has addressed the analysis of function pointers in C [48, 144, 147] as well as virtual method calls in C++ [3, 12, 30, 161] and Java [71, 72, 166, 184, 189]. Reflection, however, has mostly been neglected.

Reflection in Java allows the developer to perform runtime actions given the descriptions of the objects involved: one can create objects given their class names, call methods by their name, and access object fields given their name [54]. Because names of methods to be invoked can be supplied by the user, especially in the presence of dynamic class loading, precise static construction of a call graph is generally undecidable. Even if we assume that all classes that may be used are available for analysis, without placing *any restrictions* on the targets of reflective calls, a sound (or conservative) call graph would be prohibitively large.

Many projects that use static analysis for optimization, error detection, and other

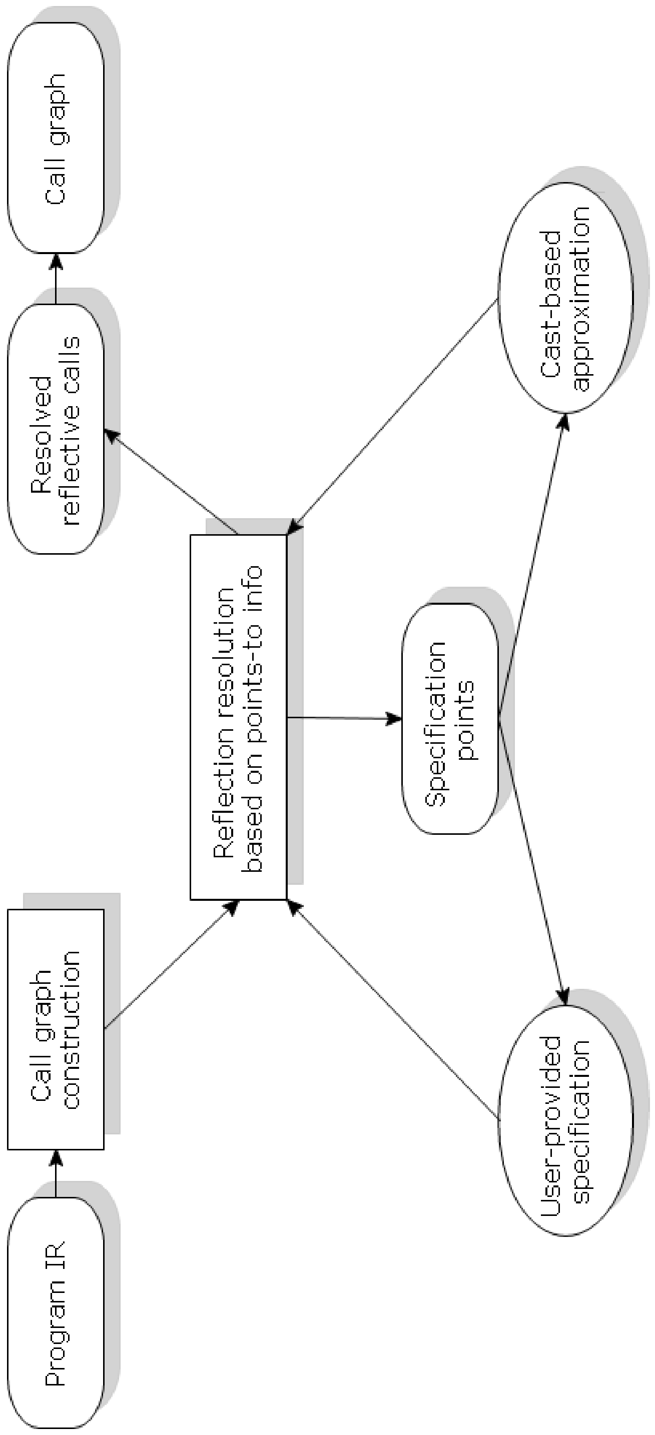

Program IR
Call graph construction
Reflection resolution based on points-to info
Resolved reflective calls
Call graph
Specification points
User-provided specification
Cast-based approximation

Figure 4.1: Architecture of our static analysis framework for call graph construction in the presence of reflection.

Retrieving `Class` or `Constructor` objects:

`Class`	`java.lang.Class.forName(String className)`
`Class`	`java.lang.Class.forName(String name, boolean initialize, ClassLoader loader)`
`Class`	`java.lang.Object.getClass()`
`Constructor[]`	`java.lang.Class.getConstructors()`
`Constructor`	`java.lang.Class.getConstructor(Class[] args)`

Creating new objects:

`Object`	`java.lang.Class.newInstance()`
`Object`	`java.lang.reflect.Constructor.newInstance(Object[] initargs)`

Figure 4.2: Java API methods for reflective object creation.

purposes ignore the use of reflection, which makes static analysis tools *incomplete* because some parts of the program may not be included in the call graph and potentially *unsound*, because some operations, such as reflectively invoking a method or setting an object field, are ignored. Others require the user to specify the methods invoked reflectively [188].

In recent years there has been an upsurge of interest in the use of static analysis for error detection. Our research is motivated by the practical need to improve the coverage of static error detection tools [111, 129, 167, 201]. The success of such tools in Java is predicated upon having a call graph available to the error detection tool. Unless reflective calls are interpreted, the tools run the danger of only analyzing a small portion of the available code and giving the developer a false sense of security when no bugs are reported. Moreover, when static results are used to reduce runtime instrumentation, all parts of the application that are used at runtime *must* be statically analyzed. While finding *some* bugs is valuable, a tool that claims to find *all* possible bugs of a particular kind provides a much stronger guarantee: a software system for which no errors are statically reported in known to be error-free.

A recent paper by Hirzel, Diwan, and Hind proposes the use of dynamic instrumentation to collect the reflection targets discovered at runtime [85]. They use this information to extend Andersen's context-insensitive, inclusion-based pointer analysis for Java into an online algorithm [5]. Reflective calls are generally used to offer a choice in the application control flow, and a dynamic application run typically includes only several of all the possibilities. However, analyses used for static error detection and optimization often require a *full* call graph of the program in order to achieve complete coverage.

In this paper we present a static analysis algorithm that uses points-to information to determine the targets of reflective calls. Often the targets of reflective calls can be determined precisely by analyzing the flow of strings that represent class names throughout the program. This allows us to precisely resolve many reflective calls and add them to the call graph. However, in some cases reflective call targets may depend on user input and require user-provided specifications for the call graph to be determined. Our algorithm determines all *specification points* — places in the program where user-provided specification is needed to determine reflective targets. The user is given the option to provide a specification and our call graph is complete with respect to the specifications provided [189].

Because providing reflection specifications can be time-consuming and error-prone, we also provide a conservative, albeit sometimes imprecise, approximation of targets of reflective calls by analyzing how type casts are used in the program. A common coding idiom consists of casting the result of a call to `Class.newInstance` used to create new objects to a more specific type before the returned object can be used. Relying on cast information allows us to produce a conservative call graph approximation without requiring user-provided reflection specifications in most cases. A flow diagram summarizing the stages of our analysis is shown in Figure 4.1.

Our reflection resolution approach hinges on three assumptions about the use of

Retrieving `Method` objects:

```
Method    java.lang.Class.getDeclaredMethod(String name, Class[] parameterTypes)
Method[]  java.lang.Class.getDeclaredMethods()
Method[]  getMethods()
```

Calling methods:

```
Object    java.lang.reflect.Method.invoke(Object obj, Object[] args)
```

Figure 4.3: Java API methods for method invocation.

Retrieving `Field` objects:

`Field`	`getField(String name)`
`Field[]`	`getFields()`
`Field`	`getDeclaredField(String name)`
`Field[]`	`getDeclaredFields()`

Accessing field values: getter and setter methods for objects and primitive types:

`Object`	`java.lang.Reflect.Field.get(Object obj)`
`byte`	`java.lang.Reflect.Field.getByte(Object obj)`
`char`	`java.lang.Reflect.Field.getChar(Object obj)`
`int`	`java.lang.Reflect.Field.getInt(Object obj)`
	...
`void`	`java.lang.Reflect.Field.set(Object obj, Object value)`
`void`	`java.lang.Reflect.Field.setByte(Object obj, byte b)`
`void`	`java.lang.Reflect.Field.setChar(Object obj, char c)`
`void`	`java.lang.Reflect.Field.setInt(Object obj, int i)`

Figure 4.4: Java API methods for field access and manipulation.

reflection: (a) all the class files that may be accessed at runtime are available for analysis; (b) the behavior of `Class.forName` is consistent with its API definition in that it returns a class whose name is specified by the first parameter, and (c) cast operations that operate on the results of `Class.newInstance` calls are correct. In rare cases when no cast information is available to aid with reflection resolution, we report this back to the user as a situation requiring specification.

4.1.1 Chapter Contributions

This chapter makes the following contributions:

- We present a case study of common uses of reflection in large modern Java systems. This study shows the importance of handling reflection in call graph construction.
- We formulate a set of natural assumptions that hold in most Java applications and make the use of reflection amenable to static analysis.

- We propose a call graph construction algorithm that uses points-to information about strings used in reflective calls to statically find potential call targets. When reflective calls cannot be fully "resolved" at compile time, our algorithms determines a set of specification points — places in the program that require user-provided specification to resolve reflective calls.

- As an alternative to having to provide a reflection specification, we propose an algorithm that uses information about type casts in the program to statically approximate potential targets of reflective calls.

- We provide an extensive experimental evaluation of our analysis approach based on points-to results by applying it to a suite of six large open-source Java applications consisting of more than 600,000 lines of code combined. We evaluate how the points-to and cast-based analyses of reflective calls compare to a local intra-method approach. While all these analyses find at least one constant target for most `Class.forName` call sites, they only moderately increase the call graph size. However, the conservative call graph obtained with the help of a user-provided specification results is a call graph than is almost 7 times as big as the original. We assess the amount of effort required to come up with a specification and how cast-based information can significantly reduce the specification burden placed on the user.

4.1.2 Chapter Organization

The rest of the chapter is organized as follows. In Section 4.2, we provide background information about the use of reflection in Java. In Section 4.3 we show some of the common usage idioms that justify the need for reflection analysis. In Section 4.4, we lay out the simplifying assumptions made by our static analysis. In Sections 4.5 we describe our analysis approach. Section 4.6 provides a comprehensive experimental evaluation. In Section 4.7 concludes this chapter. Related work is described in Section 7.6.

4.2 Overview of Reflection in Java

In this section we first informally introduce the reflection APIs in Java and then show some characteristic ways in which they are used in large Java applications.

4.2.1 Reflection APIs in Java

The most typical use of reflection by far is for creating new objects given the object class name. The most common usage idiom for reflectively creating an object is shown in Figure 4.5. In the rest of this section, we fully describe reflective APIs that Java provides for creating objects, invoking methods, and reading and writing to data structures at runtime. There are also read-only API methods that are used for *runtime discovery*; for example, an application can check if a certain method exists before trying to invoke it.

Obtaining `Class` Objects

Obtaining a class given its name is most typically done using a call to one of the static functions `Class.forName(String, ...)` and passing the class name as the first parameter. We should point out that while `Class.forName` is the most common way to obtain a class given its name, it may not be the only method for doing so. An application may define a native method that implements the same functionality. The same observation applies to other standard reflective API methods.

The `.class` construct is syntactic sugar that is translated by the compiler down into basic calls to `Class.forName`. The translation is somewhat different depending on the version of the compiler. For example, when `T.class` is translated, Sun's version of `javac` in JDK 1.4 produces bytecode shown in Figure 4.6. In this case, method

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

Figure 4.5: Typical use of reflection to create new objects.

```
static java.lang.Class class$test$T;

   ...
   0:   getstatic       #7; //Field class$test$T:Ljava/lang/Class;
   3:   ifnonnull       18
   6:   ldc             #8; //String test$T
   8:   invokestatic    #9; //class$:(Ljava/lang/String;)Ljava/lang/Class;
   11:  dup
   12:  putstatic       #7; //Field class$test$T:Ljava/lang/Class;
   15:  goto            21
   18:  getstatic       #7; //Field class$test$T:Ljava/lang/Class;
   21:  astore_1
   22:  getstatic       #10; //Field java/lang/System.out:LPrintStream;
   25:  new             #11; //class StringBuffer
   28:  dup
   29:  invokespecial   #12; //StringBuffer."<init>":()V
   32:  ldc             #13; //String c:
   34:  invokevirtual   #14; //StringBuffer.append:(LString;)LString
   37:  aload_1
   38:  invokevirtual   #15; //StringBuffer.append:(LObject;)LString
   41:  invokevirtual   #16; //StringBuffer.toString:()LString;
   44:  invokevirtual   #17; //PrintStream.println:(LString;)V
   47:  return

static java.lang.Class class$(java.lang.String);
  Code:
   0:   aload_0
   1:   invokestatic    #1; //Class.forName:(Ljava/lang/String;)LClass;
   4:   areturn
   5:   astore_1
   6:   new             #3; //class NoClassDefFoundError
   9:   dup
   10:  aload_1
   11:  invokevirtual   #4; //ClassNotFoundException.getMessage:()LString;
   14:  invokespecial   #5; //NoClassDefFoundError."<init>":(LString;)V
   17:  athrow
  Exception table:
   from   to  target type
     0     4     5   Class java/lang/ClassNotFoundException
```

Figure 4.6: Interpretation of .class in JDK version 1.4 and below.

class$ that takes a class name and returns the class returned by `Class.forName` is generated by the compiler. The result of the call is stored in field `class$test$T`. The same compiler in JDK 1.5 takes a more efficient approach that results in a much shorter bytecode sequence:

```
ldc_w   #2; //class test$T
astore_1
```

In this case, the `Class` object is loaded from a constant pool. The analysis described here handles the JDK 1.4 interpretation; supporting the JDK 1.5 interpretation requires a simple extension of our algorithm to reflect the creation of the constant pool.

Reflective Object Creation

Object creation APIs in Java provide a way to programmatically create objects of a class, whose name is provided at runtime; parameters of the object constructor can be passed in as necessary. Relevant Java API methods are summarized in Figure 4.2. Creating an object with an empty constructor is achieved through a call to `newInstance` on the appropriate `java.lang.Class` object, which provides a runtime representation of a class.

While methods `Class.forName` and `Class.newInstance` represent the majority of uses of reflection in real-life software systems, Java also provides ways to reflectively invoke a method given its name and to access the value of an object field at runtime, as described below [54].

Reflective Method Invocation

Methods are obtained from a `Class` object by supplying the method signature or by iterating through the array of `Methods` returned by one of `Class` functions. `Methods` are subsequently invoked by calling `Method.invoke`. The complete list of relevant API functions is summarized in Figure 4.3.

Reflective Field Access

Fields of Java runtime objects can be read and written at runtime. Calls to `Field.get` and `Field.set` can be used to get and set fields containing objects. Additional methods are provided for fields of primitive types. (All Java primitive types are supported, we limit the list in Figure 4.4 to several representative ones only.)

4.3 Use of Reflection: Case Studies

In this Section, we describe some of the common usage patterns for reflection found in large Java systems. We identified these patterns by studying large Java applications downloaded from SourceForge; more details on these applications can be found in Section 4.6. In addition to describing each use case, we show why statically resolving reflection is important.

4.3.1 Specifying Application Extensions

Many large applications support plugins, which are usually detected by the application upon startup by either reading a specification file or looking for files in a specific directory. For example, `columba`, an open-source email client parses an XML specification file to determine which plugins to instantiate, as shown in Figure 4.7. A similar scheme is supported by `jedit`, which also supports high levels of customization and downloadable plugins.

Application servers such as Apache Tomcat, use similar schemes where they retrieve plugin descriptions from files or by traversing a predefined directory $\mathtt{WEB}-\mathtt{INF}$ [25]. Clearly, static analysis needs to be aware of these application extensions. If reflective calls are not properly resolved, most of the plugins or Web applications in the case of Tomcat would be completely missing from the call graph. However, this represents a case where without "hints" from the user static analysis cannot determine which plugins to analyze.

4.3.2 Custom-made Object Serialization Scheme

Often objects are reflectively created based on a specification that is passed to the application.

Example 4.1. Reflection is used by an open-source genetic algorithm library, `jgap`,

```
public void addHandlers(String path) {
        XmlIO xmlFile = new XmlIO(DiskIO.getResourceURL(path));
        xmlFile.load();

        XmlElement list = xmlFile.getRoot().getElement("handlerlist");
        Iterator it = list.getElements().iterator();
        while (it.hasNext()) {
            XmlElement child = (XmlElement) it.next();
            String id = child.getAttribute("id");
            String clazz = child.getAttribute("class");

            AbstractPluginHandler handler = null;
            try {
                Class c = Class.forName(clazz);
                handler = (AbstractPluginHandler) c.newInstance();
                registerHandler(handler);
            } catch (ClassNotFoundException e) {
                if (Main.DEBUG) e.printStackTrace();
            } catch (InstantiationException e1) {
                if (Main.DEBUG) e1.printStackTrace();
            } catch (IllegalAccessException e1) {
                if (Main.DEBUG) e1.printStackTrace();
            }
        }
    }
```

Figure 4.7: Creating objects reflectively based on an XML specification.

```
String geneClassName = thisGeneElement.
           getAttribute(CLASS_ATTRIBUTE);

Gene thisGeneObject = (Gene) Class.forName(
             geneClassName).newInstance();
```

Figure 4.8: Creating objects reflectively based on an XML specification.

```
try {
    Class macOS  = Class.forName("gruntspud.standalone.os.MacOSX");
    Class argC[] = {ViewManager.class};
    Object arg[] = {context.getViewManager()};
    Method init = macOS.getMethod("init", argC);
    Object obj  = macOS.newInstance();
    init.invoke(obj, arg);
} catch (Throwable t) {
    // not on macos
}
```

Figure 4.9: Calling a method if it is present on the runtime platform.

```
    Method m = c.getMethod("clone", null);
    if (Modifier.isPublic(m.getModifiers())) {
        try {
            result = m.invoke(object, null);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Figure 4.10: Checking if a method is present before calling it.

```
try {
    //  Test for being run under JDK 1.4+
    Class.forName("javax.imageio.ImageIO");
    //  Test for JFreeChart being compiled
    //  under JDK 1.4+
    Class.forName("org.jfree.chart.encoders.SunPNGEncoderAdapter");
} catch (ClassNotFoundException e) {
    ...
}
```

Figure 4.11: Using reflection to circumventing JDK inconsistencies.

```
    Method getVersionMethod =
        Class.forName("org.columba.core.main.ColumbaVersionInfo").
            getMethod("getVersion", new Class[0]);

    return (String) getVersionMethod.invoke(null,new Object[0]);
```

Figure 4.12: Using a method that is not available at compile time.

```
public JDBCCategoryDataset(String url, String driverName,
                           String user, String passwd)
    throws ClassNotFoundException, SQLException
{
    Class.forName(driverName);
    this.connection = DriverManager.getConnection(url, user, passwd);
}
```

Figure 4.13: Using `Class.forName` for its side-effects.

```
fieldSysPath = ClassLoader.class.getDeclaredField("sys_paths");
fieldSysPath.setAccessible(true);

if (fieldSysPath != null) {
    fieldSysPath.set(System.class.getClassLoader(), null);
}
```

Figure 4.14: Circumventing static type checking to set a field.

to implement a customized serialization scheme. Information on genes is saved in an XML file and then later loaded to create runtime data structures. The names of the classes to be created are read from an XML element on lines 1–2 in Figure 4.8 and the objects are created on lines 4–5. □

Like the plugin examples above, this example demonstrates the need for user-provided specifications of reflective targets [189] when the strings on which reflective calls depend are not constant.

4.3.3 Improving Portability Using Reflection

While many Java applications are fully platform-independent, there are often subtle reasons to use platform-specific code, especially in large systems.

Example 4.2. Reflection is used in `gruntspud`, an open-source graphical CVS client, to improve code portability across different platforms. As shown in the code excerpt in Figure 4.9, call of method `init` is executed *only* if the call to `Class.forName` in the `try` clause succeeds when the Mac-OS-specific class is available. □

Similarly, sometimes applications check if certain methods are available before calling them:

Example 4.3. A generic cloning routine in `jfreechart` checks that `clone` is available and declared to be `public` before attempting to call it, as shown in Figure 4.10. The call is attempted only if the method is `public`. □

Platform-dependent features are not the only reason to use reflection for the purpose of introspection. The behavior of the program can also differ depending on the JDK version being used.

Example 4.4. The code in Figure 4.11 illustrates another use of reflection to get around incompatibilities in the JDK implementations across different distributions. The code conditionally creates an instance of `SunPNGEncoderAdapter` if `jfreechar` is used with a JDK version 1.4 and above. □

Examples 2 — 4 illustrate an inherent weakness of dynamic analysis that manifests itself when it comes to platform-specific code. Only a subset of the code in an application is executed on any particular platform. Static techniques, on the other hand, can analyze parts of code intended to be executed on different platforms all at once. If we want to detect subtle platform-specific errors that are hard to reproduce at runtime, obtaining a full call graph of the application requires reflection resolution.

4.3.4 Code Unavailable Until Deployment

Reflection is also used to examine information that does not exist at compile time and only becomes available after the application is deployed.

Example 4.5. The code from `columba` in Figure 4.12 invokes method `getVersion` of class `org.columba.core.main.ColumbaVersionInfo`. Upon examining the code, we found that this class is not created until after the application is deployed, which is when the version information becomes available. □

As described in Section 4.5.3, to make classes generated at deployment time available to our analysis, our techniques collect information on all classes *after* application deployment. Example 5 is a specific case of a more general Java design pattern, in which interface types are used and their implementations are substituted in a manner that is deployment-specific. Unless reflection is resolved, all objects of the interface types will lack an implementation that can be statically analyzed.

4.3.5 Using `Class.forName` for its Side-effects

The call to `Class.forName` has the additional effect of calling the class constructor of the class being referenced. Occasionally, the result of the call is ignored and the call is used as a convenient way to invoke the class constructor. This coding idiom is commonly used to initialize database drivers as shown in a code excerpt extracted from `jfreechart` in Figure 4.13.

4.3.6 Getting Around Static Type Checking

While this is relatively uncommon, reflection makes it possible to circumvent the standard Java type system.

Example 4.6. As shown in the code snippet in Figure 4.14 extracted from `columba`, reflection is used to reset the system library paths of the default class loader by setting field `sys_paths` to `null`. Since the field is non-public, the accessibility flag of the field is first reset on line 2 before assigning to the field on lines 5. □

If we fail to take into account methods that assign to fields of objects when constructing the program representation, the resulting representation will be incomplete.

4.3.7 Providing a Built-in Interpreter

On occasion, a very wide set of classes may be returned by a reflective calls, as shown below.

Example 4.7. One of our benchmark applications, `jedit`, contains an embedded BeanShell, a Java source interpreter used to write editor macros [155]. Within the BeanShell interpreted, one of the calls to `Class.forName` takes type parameters extracted from the Bean shell macros. □

Clearly, ignoring the targets of the `Class.forName` call in this case leads to the code within the macro file not being analyzed. But this example also reveals a key difficulty: the set of macros is hardly static. New macros can be downloaded or written, so the approximation of the reflective targets is only valid with respect to a specific application configuration.

4.4 Assumptions About Reflection

This section presents assumptions we make in our static analysis for resolving reflection in Java programs. We believe that these assumptions are quite reasonable and hold for many real-life Java applications.

The problem of precisely determining the classes that an application may access is undecidable. Furthermore, for applications that access the network, the set of classes that may be accessed is *unbounded*: we cannot possibly hope to analyze all classes that the application may conceivably download from the net and load at runtime. Programs can also dynamically generate classes to be subsequently loaded. Our analysis assumes a closed world, as defined below.

Assumption 4.1 Closed world.
We assume that only classes reachable from the class path at analysis time can be used by the application at runtime.

In the presence of user-defined class loaders, it is impossible to statically determine the behavior of function `Class.forName`. If custom class loaders are used, the behavior of `Class.forName` can change; it is even possible for a malicious class loader to return completely unrelated classes in response to a `Class.forName` call. The following assumption allows us to interpret calls to `Class.forName`. We assume that the behavior of `Class.forName` is consistent with the API declaration even when custom class loaders are used, which postulates that:

> *Given the fully qualified name for a class or interface (in the same format returned by* `getName`*) this method attempts to locate, load, and link the class or interface.*

Assumption 4.2 Well-behaved class loaders.
The name of the class returned by a call to `Class.forName(className)` *equals* `className`*.*

To check the validity of this assumption, we have instrumented large applications to observe the behavior of `Class.forName`; we have never encountered a violation of this assumption. Finally, we introduce an assumption that allows us to leverage type cast information contained in the program to constrain the targets of reflective calls.

Assumption 4.3 Correct casts.
Type cast operations that always operate on the result of a call to `newInstance` *are correct; they will always succeed without throwing a* `ClassCastException`*.*

We believe this to be a valid practical assumption: while it is possible to have casts that fail, causing an exception that is caught so that the instantiated object can be used afterwards, we have not seen such cases in practice. Typical `catch` blocks around such casts lead to the program terminating with an error message.

4.5 Analysis of Reflection

In this section, we present techniques for resolving reflective calls in a program. Our analysis consists of the following three steps:

1. We use a sound points-to analysis to determine all the possible sources of strings that are used as class names. Such sources can either be constant strings or derived from external sources. The pointer analysis-based approach *fully resolves* the targets of a reflective call if constant strings account for all the possible sources. We say that a call is *partially resolved* if the sources can be either constants or inputs and *unresolved* if the sources can only be inputs. Knowing which external sources may be used as class names is useful because users can potentially specify all the possible values; typical examples are return results of file read operations. We refer to program points where the input strings are defined as *specification points*.

2. Unfortunately the number of specification points in a program can be large. Instead of asking users to specify the values of every possible input string, our second technique takes advantage of casts, whenever available, to determine a conservative approximation of targets of reflective calls *that are not fully resolved.* For example, as shown in Figure 4.5, the call to `Class.newInstance`, which returns an `Object`, is always followed by a cast to the appropriate type before the newly created object can be used. Assuming no exception is raised,

we can conclude that the new object must be a subtype of the type used in the cast, thus restricting the set of objects that may be instantiated.

3. Finally, we rely on user-provided specification for the remaining set of calls — namely calls whose source strings are not all constants — in order to obtain a conservative approximation of the call graph.

We start by describing the call graph discovery algorithm in Section 4.5.1 as well as how reflection resolution fits in with call graph discovery. Section 4.5.2 presents a reflection resolution algorithm based on pointer analysis results. Finally, Section 4.5.3 describes our algorithm that leverages type cast information for conservative call graph construction without relying on user-provided specifications.

4.5.1 Call Graph Discovery

Our static techniques to discover reflective targets are integrated into a context-insensitive points-to analysis that discovers the call graph on the fly [205]. As the points-to analysis finds the pointees of variables, type information of these pointees is used to resolve the targets of virtual method invocations, increasing the size of the call graph, which in turn is used to find more pointees. Our analysis of reflective calls further expands the call graph, which is used in the analysis to generate more points-to relations, leading to bigger call graphs. The discovery algorithm terminates when a fixpoint is reached and no more call targets or points-to relations can be found.

By using a points-to analysis to discover the call graph, we can obtain a more accurate call graph than by using a less precise technique such as class hierarchy analysis CHA [46] or rapid type analysis RTA [11]. We use a context-insensitive version of the pointer analysis for this purpose because context sensitivity does not seem to substantially improve the accuracy of the call graph [205, 71] and the context-insensitive version is substantially faster.

4.5.2 Pointer Analysis for Reflection

This section describes how we leverage pointer analysis results to resolve calls to `Class.forName` and track `Class` objects. This can be used to discover the types of objects that can be created at calls to `Class.newInstance`, along with resolving reflective method invocations and field access operations. Pointer analysis is also used to find specification points: external sources that propagate string values to the first argument of `Class.forName`.

Reflection and Points-to Information

The programming idiom that motivated the use of points-to analysis for resolving reflection was first presented in Figure 4.5. This idiom consists of the following steps:

1. Obtain the name of the class for the object that needs to be created.
2. Create a `Class` object by calling the static method `Class.forName`.
3. Create the new object with a call to `Class.newInstance`.
4. Cast the result of the call to `Class.newInstance` to the necessary type in order to use the newly created object.

When interpreting this idiom statically, we would like to "resolve" the call to `Class.newInstance` in step 3 as a call to the default constructor `T()`. However, analyzing even this relatively simple idiom is nontrivial.

The four steps shown above can be widely separated in the code and reside in different methods, classes, or jar libraries. The `Class` object obtained in step 2 may be passed through several levels of function calls before being used in step 3. Furthermore, the `Class` object can be deposited in a collection to be later retrieved in step 3. The same is true for the name of the class created in step 1 and used later in step 2. To determine how variables `className`, `c`, `o`, and `t` defined and used in steps 1–4 may be related, we need to know what runtime objects they may be referring to: a problem addressed by *points-to* analysis. Point-to analysis computes which objects each program variable may refer to.

Resolution of `Class.newInstance` of `Class.forName` calls is not the only thing made possible with points-to results: using points-to analysis, we also track `Method`, `Field`, and `Constructor` objects. This allows us to correctly resolve reflective method invocations and field accesses. Reflection is also commonly used to invoke the class constructor of a given class via calling `Class.forName` with the class name as the first argument. We use points-to information to determine potential targets of `Class.forName` calls and add calls to class constructors of the appropriate classes to the call graph.

Datalog Relations Used for Reflection Resolution

As described in Section 3.3.1, the input program is represented as a set of relations[1]. Several additional relations are used for the process of reflection resolution.

string2class: $H \times T$. *string2class*(s, t) means that string constant s is the string representation of the name of type t.

string2method: $H \times M$. *string2method*(s, m) means that string constant s is the string representation of the name of method m.

string2field: $H \times F$. *string2field*(s, f) means that string constant s is the string representation of the name of field f.

Finally, an auxiliary relation *freshi2h* is used for reflection resolution to make "fresh" heap allocation sites:

freshi2h: $I \times H$. *freshi2h*(i, h) means that a *freshly created* allocation site h corresponds to the result of the method call at i.

Pointer Analysis Results

In addition to input relations, relations

$$\textit{points-to} : V \times H$$

[1] In the rest of this chapter, we shall use the context-insensitive version of the *points-to* relation, *points-to*$(v, h) : V \times H$.

and

$$\textit{hpoints-to} : H \times F \times H$$

are computed as a result of pointer analysis. Pointer analysis computation is integrated with the process of call graph discovery so that new points-to facts are introduced as more and more of the call graph is analyzed.

In the rest of this chapter, we shall use the context-insensitive version of the $\textit{points-to}(v \in V, h \in H)$ relation. While a slight increase in precision may be gained by using context-sensitive results, the approach to call graph numbering proposed by Whaley and Lam requires the call graph to be known in advance [205].

Basics of Reflection Resolution Using Points-To Information

The algorithm for computing targets of reflective calls is naturally expressed in terms of Datalog queries. Below we define Datalog rules to resolve targets of `Class.newInstance` and `Class.forName` calls. Handling of constructors, methods, and fields proceed similarly, as described in Sections 4.5.2–4.5.2. To disambiguate relations introduced for reflection resolution from input relations, we use Java identified naming conventions for the former.

To compute reflective targets of calls to `Class.newInstance`, we define two Datalog relations. Relation *classObjects* contains pairs $\langle i, t\rangle$ of invocations sites $i \in I$ calling `Class.forName` and types $t \in T$ that may be returned from the call. We define *classObjects* using the following Datalog rule:

$$\begin{aligned} \textit{classObjects}(i, t) \ :- \ & \textit{call}(i, \text{“}\texttt{Class.forName}\text{”}), \\ & \textit{actual}(i, 1, v), \textit{points-to}(v, s), \\ & \textit{string2class}(s, t). \end{aligned}$$

The Datalog rule for *classObjects* reads as follows. Invocation site i returns an object of type t if the call graph relation *call* contains an edge from i to “`Class.forName`”, parameter 1 of i is v, v points to s, and s is a string that represents the name of type t.

Relation *newInstanceTargets* contains pairs $\langle i, t\rangle$ of invocation sites $i \in I$ calling

`Class.newInstance` and classes $t \in T$ that may be reflectively invoked by the call. The Datalog rule to compute *newInstanceTargets* is:

$$\begin{aligned} \mathit{newInstanceTargets}(i,t) \ :- \ & \mathit{call}(i, \text{``}\texttt{Class.newInstance}\text{''}), \\ & \mathit{actual}(i,0,v), \mathit{points\text{-}to}(v,c), \\ & \mathit{points\text{-}to}(v_c,c), \mathit{ret}(i_c,v_c), \\ & \mathit{classObjects}(i_c,t). \end{aligned}$$

The rule reads as follows. Invocation site i returns a new object of type t if the call graph relation *call* contains an edge from i to `Class.newInstance`, parameter 0 of i is v, v is aliased to a variable v_c that is the return value of invocation site i_c, and i_c returns type t. Targets of `Class.forName` calls are resolved and calls to the appropriate class constructors are added to the invocation relation *call*:

$$\mathit{call}(i,m) \ :- \ \mathit{classObjects}(i,t), m = t + \text{``}.<\texttt{clinit}>\text{''}.$$

(The "+" sign indicates string concatenation.) Similarly, having computed relation *newInstanceTargets*(i,t), we add these reflective call targets invoking the appropriate type constructor to the call graph relation *call* with the rule below:

$$\mathit{call}(i,m) \ :- \ \mathit{newInstanceTargets}(i,t), m = t + \text{``}.<\texttt{init}>\text{''}.$$

In Section 4.5.2 we cover other ways to perform reflective operations.

Handling Reflective Constructor Calls: `Constructor` Objects

Another technique of reflective object creation is to use `Class.getConstructor` to get a `Constructor` object, and then calling `newInstance` on that. We define a relation *constructorTypes* that contains pairs $\langle i,t\rangle$ of invocations sites $i \in I$ calling

`Class.getConstructor` and types $t \in T$ of the type of the constructor:

$$\begin{aligned} \textit{constructorTypes}(i,t) \;:-\; & \textit{call}(i, \text{“}\texttt{Class.getConstructor}\text{”}), \\ & \textit{actual}(i,0,v), \textit{points-to}(v,h), \\ & \textit{classObjects}(h,t). \end{aligned}$$

Once we have computed *constructorTypes*, we can compute more *newInstanceTargets* as follows:

$$\begin{aligned} \textit{newInstanceTargets}(i,t) \;:-\; & \textit{call}(i, \text{“}\texttt{Class.newInstance}\text{”}), \\ & \textit{actual}(i,0,v), \textit{points-to}(v,c), \textit{points-to}(v_c,c), \\ & \textit{ret}(i_c,v_c), \textit{constructorTypes}(i_c,t). \end{aligned}$$

This rule says that invocation site i calling method “`Class.newInstance`” returns an object of type t if parameter 0 of i is v, v is aliased to the return value of invocation i_c which calls “`Class.getConstructor`”, and the call to i_c is on type t.

In a similar manner, we add support for `Class.getConstructors`, along with support for reflective field, and method accesses. The specification of these are straightforward and we do not describe them here.

Handling Reflective Invocations: `Method` Objects

Auxiliary relation *getMethod* defines two standard ways reflection API ways to reflectively invoke a method.

$$\begin{aligned} & \textit{getMethod}(\text{“}\texttt{Class.getMethod(String, Class}\text{”}). \\ & \textit{getMethod}(\text{“}\texttt{Class.getDeclaredMethod(String, Class}\text{”}). \end{aligned}$$

Relation $\textit{methodObject}(h,m)$ defines when a heap-allocated object $h \in H$ represents a method $m \in M$. Relation $\textit{resolvedInvoke}(i,m)$ defines when a method $m \in M$ can be called from an invocation site $i \in I$.

$$\begin{aligned}
\mathit{methodObject}(h, m) \quad &:- \quad \mathit{getMethod}(m_i), \mathit{call}(i, m_i), \mathit{freshi2h}(i, h),\\
&\qquad \mathit{actual}(i, 1, v), \mathit{points\text{-}to}(v, h_m),\\
&\qquad \mathit{string2method}(h_m, m).\\
\mathit{resolvedInvoke}(i, m) \quad &:- \quad \mathit{call}(i, \text{“}\texttt{Method.invoke(Object, Object[])}\text{”}),\\
&\qquad \mathit{actual}(i, 0, v), \mathit{points\text{-}to}(v, h), \mathit{methodObject}(h, m).
\end{aligned}$$

Finally, input relations are updated with the results of method invocation resolution to represent the flow of data through parameters and return values:

$$\begin{aligned}
\mathit{assign}(v_1, v_2) \quad &:- \quad \mathit{resolvedInvoke}(i, m),\\
&\qquad \mathit{formal}(m, 0, v_1), \mathit{actual}(i, 1, v_2).\\
\mathit{assign}(v_1, v_2) \quad &:- \quad \mathit{resolvedInvoke}(i, m), \mathit{ret}(i, v_1), \mathit{mret}(m, v_2).\\
\mathit{points\text{-}to}(v, h) \quad &:- \quad \mathit{resolvedInvoke}(i, m), \mathit{formal}(m, z, v), z > 0,\\
&\qquad \mathit{actual}(i, 2, v_2), \mathit{points\text{-}to}(v_2, h_2),\\
&\qquad \mathit{hpoints\text{-}to}(h_2, \text{“}\texttt{null}\text{”}, h).
\end{aligned}$$

Handling Reflective Field Accesses: `Field` Objects

Resolution of `Field` objects is similar to `Method`s. Auxiliary relation *getField* defines two standard ways reflection API ways to reflectively access a field.

$$\begin{aligned}
&\mathit{getField}(\text{“}\texttt{Class.getField(String)}\text{”}).\\
&\mathit{getField}(\text{“}\texttt{Class.getDeclaredField(String)}\text{”}).
\end{aligned}$$

Relation *fieldObject*(h, f) defines when a heap-allocated object $h \in H$ represents a field $f \in F$.

$$\begin{aligned}
\mathit{fieldObject}(h, f) \quad &:- \quad \mathit{getField}(m_f), \mathit{call}(i, m_f),\\
&\qquad \mathit{freshi2h}(i, h), \mathit{actual}(i, 1, v), \mathit{points\text{-}to}(v, h_f),\\
&\qquad \mathit{string2field}(h_f, f).
\end{aligned}$$

Finally, *load* and *store* relations are updated to represent the newly discovered reflective field accesses:

$$\begin{aligned}
\mathit{store}(v_1, f, v_2) \quad &:- \quad \mathit{call}(i, \text{“}\texttt{Field.set(Object,Object)}\text{”}),\\
&\qquad \mathit{actual}(i, 0, v), \mathit{points\text{-}to}(v, h), \mathit{fieldObject}(h, f),\\
&\qquad \mathit{actual}(i, 1, v_1), \mathit{actual}(i, 2, v_2).\\
\mathit{load}(v_1, f, v_2) \quad &:- \quad \mathit{call}(i, \text{“}\texttt{Field.get(Object,Object)}),\\
&\qquad \mathit{actual}(i, 0, v), \mathit{points\text{-}to}(v, h), \mathit{fieldObject}(h, f),\\
&\qquad \mathit{actual}(i, 1, v_1), \mathit{ret}(i, v_2).
\end{aligned}$$

Notice that because our relations describing the input program only represent *objects* and effectively ignore primitive values, we do not need to model other field accessors such as `Field.getByte`/`Field.setByte`, etc. listed in Figure 4.4.

Specification Points and User-Provided Specifications

Using a points-to analysis also allows us to determine, when a non-constant string is passed to a call to `Class.forName`, the *provenance* of that string. The *provenance* of a string is in essence a backward data slice showing the flow of data to that string. Provenance allows us to compute *specification points* — places in the program where external sources are read by the program from a configuration file, system properties, etc. For each specification point, the user can provide values that may be passed into the application.

Our implementation accepts specification files that contain a simple textual map of a specification point to the constant strings it can generate. A specification point is represented by a method name, bytecode offset, and the relevant line number. An example of a specification file is shown in Figure 4.15.

We compute provenance by propagating through the assignment relation *assign*, aliased loads and stores, and string operations. To make the specification points as close to external sources as possible, we perform a simple analysis of strings to do backward propagation through string concatenation operations. For brevity, we only list the `StringBuffer.append` method used by the Java compiler to expand string

```
loadImpl() @ 43 InetAddress.java:1231       => java.net.Inet4AddressImpl
loadImpl() @ 43 InetAddress.java:1231       => java.net.Inet6AddressImpl
...
lookup() @ 86 AbstractCharsetProvider.java:126 => sun.nio.cs.ISO_8859_15
lookup() @ 86 AbstractCharsetProvider.java:126 => sun.nio.cs.MS1251
...
tryToLoadClass() @ 29 DataFlavor.java:64       => java.io.InputStream
...
```

Figure 4.15: A fragment of a specification file accepted by our system. A string identifying a call site to `Class.forName` is mapped to a class name that that call may resolve to.

concatenation operations here; other string operations work in a similar manner. The following rules for relation *leadsToForName* detail provenance propagation:

$$
\begin{array}{lll}
\mathit{leadsToForName}(v, i) & :- & \mathit{call}(i, \text{“}\texttt{Class.forName}\text{”}), \mathit{actual}(i, 1, v). \\
\mathit{leadsToForName}(v_2, i) & :- & \mathit{leadsToForName}(v_1, i), \mathit{assign}(v_1, v_2). \\
\mathit{leadsToForName}(v_2, i) & :- & \mathit{leadsToForName}(v_1, i), \\
 & & \mathit{load}(v_3, f, v_1), \mathit{points\text{-}to}(v_3, h_3), \\
 & & \mathit{points\text{-}to}(v_4, h_3), \mathit{store}(v_4, f, v_2). \\
\mathit{leadsToForName}(v_2, i) & :- & \mathit{leadsToForName}(v_1, i), \mathit{ret}(i_2, v_1), \\
 & & \mathit{call}(i_2, \text{“}\texttt{StringBuffer.append}\text{”}), \mathit{actual}(i_2, 0, v_2). \\
\mathit{leadsToForName}(v_2, i) & :- & \mathit{leadsToForName}(v_1, i), \mathit{ret}(i_2, v_1), \\
 & & \mathit{call}(i_2, \text{“}\texttt{StringBuffer.append}\text{”}), \mathit{actual}(i_2, 1, v_2). \\
\mathit{leadsToForName}(v_2, i) & :- & \mathit{leadsToForName}(v_1, i), \mathit{actual}(i_2, 0, v_1), \\
 & & \mathit{call}(i_2, \text{“}\texttt{StringBuffer.append}\text{”}), \mathit{actual}(i_2, 1, v_2).
\end{array}
$$

To compute the specification points necessary to resolve `Class.forName` calls, we find endpoints of the *leadsToForName* propagation chains that are *not* string constants that represent class names. Relation *lhs*(v) defines when variable v is ever assigned to in the program.

$$\begin{array}{lll}
\mathit{lhs}(v) & :- & \mathit{assign}(v, \_). \\
\mathit{lhs}(v) & :- & \mathit{call}(i, \text{``}\texttt{StringBuffer.append}\text{''}), \mathit{ret}(i, v). \\
\mathit{lhs}(v) & :- & \mathit{call}(i, \text{``}\texttt{StringBuffer.append}\text{''}), \mathit{actual}(i, 0, v). \\
\mathit{lhs}(v) & :- & \mathit{call}(i, \text{``}\texttt{StringBuffer.toString}\text{''}), \mathit{ret}(i, v). \\
\mathit{lhs}(v) & :- & \mathit{call}(i, \text{``}\texttt{new StringBuffer}\text{''}), \mathit{actual}(i, 0, v). \\
\mathit{lhs}(v) & :- & \mathit{load}(v_3, f, v), \mathit{points\text{-}to}(v_3, h), \mathit{points\text{-}to}(v_2, h), \mathit{store}(v_2, f, \_).
\end{array}$$

Relation $\mathit{isTypeString}(v)$ below holds for variables v that refer to class name constants:

$$\mathit{isTypeString}(v) \quad :- \quad \mathit{points\text{-}to}(v, h), \mathit{string2class}(h, \_).$$

Finally, relation $\mathit{specPts}(v, i)$ defines when variable v is a specification point for a call to `Class.forName` at call site i:

$$\mathit{specPts}(v, i) \quad :- \quad \mathit{leadsToForName}(v, i), \neg\mathit{lhs}(v), \neg\mathit{isTypeString}(v).$$

Variables in *specPts* are often return results of calls to `System.getProperty` in the case of reading from a system property or `BufferedReader.readLine` in the case of reading from a file. By specifying the possible values at that point that are appropriate for the application being analyzed, the user can construct a complete call graph.

Dealing with Other Reflective Calls

Unfortunately, `Class.forName` calls discussed in detail in the previous section are not the only reflective calls whose results can occasionally remain unresolved without a user-provided specification. According to the definition of *newInstanceTargets* in Section 4.5.2, calls to `Class.newInstance` should all be fully resolved as long as the underlying `Class` and `Constructor` objects are fully resolved.

Similarly, according to the rules in Sections 4.5.2 and 4.5.2, method invocations and field accesses may not be fully resolved either because the underlying field or method objects are not fully resolved or because the method or field names are not fully resolved. Since the underlying objects come from `Class.forName` calls, they will be dealt with when we consider `Class.forName` resolution. However, method and field names can be not fully resolved and we need to address these cases separately by adding to the *specPts* relation.

$$\begin{array}{lll}
\mathit{leadsToInvoke}(v, i) & :- & \mathit{call}(i, \mathit{getMethod}(m)), \mathit{actual}(i, 1, v). \\
\mathit{leadsToInvoke}(v_2, i) & :- & \mathit{leadsToInvoke}(v_1, i), \mathit{assign}(v_1, v_2). \\
\mathit{leadsToInvoke}(v_2, i) & :- & \mathit{leadsToInvoke}(v_1, i), \\
 & & \mathit{load}(v_3, f, v_1), \mathit{points\text{-}to}(v_3, h_3), \\
 & & \mathit{points\text{-}to}(v_4, h_3), \mathit{store}(v_4, f, v_2). \\
\mathit{leadsToInvoke}(v_2, i) & :- & \mathit{leadsToInvoke}(v_1, i), \mathit{ret}(i_2, v_1), \\
 & & \mathit{call}(i_2, \text{“}\texttt{StringBuffer.append}\text{”}), \mathit{actual}(i_2, 0, v_2). \\
\mathit{leadsToInvoke}(v_2, i) & :- & \mathit{leadsToInvoke}(v_1, i), \mathit{ret}(i_2, v_1), \\
 & & \mathit{call}(i_2, \text{“}\texttt{StringBuffer.append}\text{”}), \mathit{actual}(i_2, 1, v_2). \\
\mathit{leadsToInvoke}(v_2, i) & :- & \mathit{leadsToInvoke}(v_1, i), \mathit{actual}(i_2, 0, v_1), \\
 & & \mathit{call}(i_2, \text{“}\texttt{StringBuffer.append}\text{”}), \mathit{actual}(i_2, 1, v_2).
\end{array}$$

Relation *isMethodString* below defines all string variables that represent method names:

$$\mathit{isMethodString}(v) \quad :- \quad \mathit{points\text{-}to}(v, h), \mathit{string2method}(h, \_).$$

Finally, we all the necessary $\langle v \in V, i \in I \rangle$ pairs to relation *specPts*:

$$\mathit{specPts}(v, i) :- \mathit{leadsToInvoke}(v, i), \neg \mathit{lhs}(v), \neg \mathit{isMethodString}(v).$$

Finding additional specification points caused by unresolved field accesses proceeds similarly. The relevant rules are shown below:

$$\begin{array}{lll}
\mathit{leadsToField}(v,i) & :- & \mathit{call}(i, \mathit{getField}(m)), \mathit{actual}(i,1,v). \\
\mathit{leadsToField}(v,i) & :- & \mathit{call}(i, \text{“}\texttt{Class.getDeclaredField}\text{”}), \mathit{actual}(i,1,v). \\
\mathit{leadsToField}(v_2,i) & :- & \mathit{leadsToField}(v_1,i), \mathit{assign}(v_1,v_2). \\
\mathit{leadsToField}(v_2,i) & :- & \mathit{leadsToField}(v_1,i), \\
 & & \mathit{load}(v_3,f,v_1), \mathit{points\text{-}to}(v_3,h_3), \\
 & & \mathit{points\text{-}to}(v_4,h_3), \mathit{store}(v_4,f,v_2). \\
\mathit{leadsToField}(v_2,i) & :- & \mathit{leadsToField}(v_1,i), \mathit{ret}(i_2,v_1), \\
 & & \mathit{call}(i_2, \text{“}\texttt{StringBuffer.append}\text{”}), \mathit{actual}(i_2,0,v_2). \\
\mathit{leadsToField}(v_2,i) & :- & \mathit{leadsToField}(v_1,i), \mathit{ret}(i_2,v_1), \\
 & & \mathit{call}(i_2, \text{“}\texttt{StringBuffer.append}\text{”}), \mathit{actual}(i_2,1,v_2). \\
\mathit{leadsToField}(v_2,i) & :- & \mathit{leadsToField}(v_1,i), \mathit{actual}(i_2,0,v_1), \\
 & & \mathit{call}(i_2, \text{“}\texttt{StringBuffer.append}\text{”}), \mathit{actual}(i_2,1,v_2). \\
 & & \\
\mathit{isFieldString}(v) & :- & \mathit{points\text{-}to}(v,h), \mathit{field2method}(h,\_). \\
 & & \\
\mathit{specPts}(v,i) & :- & \mathit{leadsToField}(v,i), \neg \mathit{lhs}(v), \neg \mathit{isFieldString}(v).
\end{array}$$

4.5.3 Reflection Resolution Using Casts

For some applications, the task of providing reflection specifications may be too heavy a burden. Fortunately, we can leverage the type cast information present in the program to automatically determine a conservative approximation of possible reflective targets. Consider, for instance, the following typical code snippet:

```
1.   Object o = c.newInstance();
2.   String s = (String) o;
```

The cast in statement 2 *post-dominates* the call to `Class.newInstance` in statement 1. This implies that all execution paths that pass through the call to `Class.newInstance` must also go through the cast in statement 2 [2]. For statement 2 not to produce a runtime exception, `o` must be a subclass of `String`. Thus, only subtypes of `String` can be created as a result of the call to `newInstance`. More generally, if the result of a `newInstance` call is *always* cast to type t, we say that only subtypes of t can be instantiated at the call to `newInstance`.

Relying on cast operations can possibly be unsound as the cast may fail, in which case, the code will throw a `ClassCastException`. Thus, in order to work, our cast-based technique relies on Assumption 4.3, the correctness of cast operations.

Preparing Subtype Information

We rely on the closed world Assumption 4.2 described in Section 4.4 to find the set of all classes possibly used by the application. The classes available at analysis time are generally distributed with the application. However, occasionally, there are classes that are generated when the application is compiled or deployed, typically with the help of an Ant script. Therefore, we generate the set of possible classes *after* deploying the application.

We pre-process all resulting classes to compute the subtyping relation $\mathit{subtype}(t_1, t_2)$ that determines when t_1 is a subtype of t_2. Preprocessing even the smallest applications involved looking at many thousands of classes because we consider all the default jars that the Java runtime system has access to. We run this preprocessing step off-line and store the results for easy access.

Using Cast Information

We integrate the information about cast operations directly into the system of constraints expressed in Datalog. We use a Datalog relation *subtype* described above, a relation *cast* that holds the cast operations, and a relation *unresolved* that holds the unresolved calls to `Class.forName`. The following Datalog rule uses cast operations applied to the return result v_{ret} of a call i to `Class.newInstance` to constrain the possible types t_c of `Class` objects c returned from calls sites i_c of `Class.forName`:

```
1.          UniqueVector voiceDirectories = new UniqueVector();
2.          for (int i = 0; i < voiceDirectoryNames.size(); i++) {
3.              Class c = Class.forName((String) voiceDirectoryNames.get(i),
4.                                                true, classLoader);
5.              voiceDirectories.add(c.newInstance());
6.          }
7.
8.          return (VoiceDirectory[]) voiceDirectories.toArray(new
9.                          VoiceDirectory[voiceDirectories.size()]);
```

Figure 4.16: A case in `freetts` where our analysis is unable to determine the type of objects instantiated on line 5 using casts.

$$\begin{aligned} \mathit{classObjects}(i_c, t) \;:-\;\; & \mathit{call}(i, \text{“}\texttt{Class.newInstance}\text{”}), \mathit{actual}(i, 0, v), \\ & \mathit{points\text{-}to}(v, c), \mathit{ret}(i, v_{ret}), \\ & \mathit{cast}(\_, t_c, v_{ret}), \mathit{subtype}(t, t_c), \\ & \mathit{unresolved}(i_c), \mathit{points\text{-}to}(v_c, c), \mathit{ret}(i_c, v_c). \end{aligned}$$

Information propagates both forward and backward — for example, casting the result of a call to `Class.newInstance` constrains the `Class` object it is called upon. If the same `Class` object is used in another part of the program, the type constraint derived from the cast will be obeyed.

Problems with Using Casts

Casts are sometimes inadequate for resolving calls to `Class.newInstance` for the following reasons. First, the cast-based approach is inherently imprecise because programs often cast the result of `Class.newInstance` to a very wide type such as `java.io.Serializable`. This produces many *potential* subclasses, only some of which are relevant in practice. Second, as our experiments show, not all calls to `Class.newInstance` have post-dominating casts, as illustrated by the following example.

Example 4.8. As shown in Figure 4.16, one of our benchmark applications, `freetts`, places the object returned by `Class.newInstance` into a vector `voiceDirectories` (line 5). Despite the fact that the objects are subsequently cast to type `VoiceDirectory[]` on line 8, intraprocedural post-dominance is not powerful enough to take this cast into account. □

Using cast information significantly reduces the need for user-provided specification in practice. While the version of the analysis that does not use cast information can be made fully sound with user specification as well, we chose to only provide a specification for the cast-based version.

4.6 Experimental Results

In this section we present a comprehensive experimental evaluation of the static analysis approaches presented in Section 4.5. In Section 4.6.1 we describe our experimental setup. Section 4.6.2 presents an overview our experimental results. Section 4.6.3 presents our baseline local reflection analysis. In Sections 4.6.4 and 4.6.5 we discuss the effectiveness of using the points-to and cast-based reflection resolution approaches, respectively. Section 4.6.6 describes the specifications needed to obtain a sound call graph approximation. Section 4.6.7 compares the overall sizes of the call graph for the different analysis versions presented in this section.

Benchmark	**Description**	**Line count**	**File count**	**Jars**	**Available classes**
`jgap`	genetic algorithms package	32,961	172	9	62,727
`freetts`	speech synthesis system	42,993	167	19	62,821
`gruntspud`	graphical CVS client	80,138	378	10	63,847
`jedit`	graphical text editor	144,496	427	1	62,910
`columba`	graphical email client	149,044	1,170	35	53,689
`jfreechart`	chart drawing library	193,396	707	6	62,885
Total		**643,028**	**3,021**	**80**	**368,879**

Figure 4.17: Summary of information about reflection benchmarks. Applications are sorted by the number of lines of code in column 3.

4.6.1 Experimental Setup

We performed our experiments on a suite of six large, widely-used open-source Java benchmark applications. These applications were selected among the most popular Java projects available on SourceForge. We believe that real-life applications like these are more representative of how programmers use reflection than synthetically created test suites, or SPEC JVM benchmarks, most of which avoid reflection altogether.

Summary of information about the applications is provided in Figure 4.17. Notice that the traditional lines of code size metric is somewhat misleading in the case of applications that rely on large libraries. Many of these benchmarks depend of massive libraries, so, while the application code may be small, the full size of the application executed at runtime is quite large. The last column of the table in Figure 4.17 lists the number of classes available by the time each application is deployed, including those in the JDK. Notice that JDK classes dominate the picture, so the difference in the number of classes across the applications is not very large.

We ran all of our experiments on an Opteron 150 machine equipped with 4GB or memory running Linux. JDK version 1.4.2_08 was used. All of the running times for our preliminary implementation were in tens of minutes, which, although a little high, is acceptable for programs of this size. Creating subtype information for use with cast-based analysis took well under a minute.

4.6.2 Evaluation Approach

We have implemented five different variations of our algorithms: None, Local, Points-to, Casts, and Sound and applied them to the benchmarks described above. None is the base version that performs no reflection resolution; Local performs a simple local analysis, as described in Section 4.6.3. Points-to and Casts are described in Sections 4.5.2 and 4.5.3, respectively.

Version Sound is augmented with a user-provided specification as described in Section 4.5.2 to make the answer conservative. Section 4.6.6 details what kind of specification we had to provide to get a conservative result. We should point out that only the Sound version provides results that are fully sound: None essentially

	None	Local			Points-to				Casts				Sound		
Benchmark	**T**	**T**	**FR**	**UR**	**T**	**FR**	**PR**	**UR**	**T**	**FR**	**PR**	**UR**	**T**	**FR**	**UR**
`jgap`	27	27	19	8	28	20	1	7	28	20	4	4	89	85	4
`freetts`	30	30	21	9	30	21	0	9	34	25	4	5	81	75	6
`gruntspud`	139	139	112	27	142	115	5	22	232	191	19	22	220	208	12
`jedit`	156	156	137	19	161	142	3	16	178	159	12	7	210	197	12
`columba`	104	105	89	16	105	89	2	14	118	101	10	7	173	167	6
`jfreechart`	104	104	91	13	104	91	1	12	149	124	10	15	169	165	4

Figure 4.18: Results of resolving `Class.forName` calls for different analysis versions.

assumes that reflective calls have no targets. Local only handles reflective calls that can be fully resolved within a single method. Points-to and Casts only provide targets for reflective calls for which either string or cast information constraining the possible targets is available; both versions unsoundly assume that the rest of the calls have no targets.

Figure 4.18 summarizes the results of resolving `Class.forName` using all five analysis versions. `Class.forName` calls represent by far the most common kind of reflective operations and we focus on them in our experimental evaluation. To reiterate the definitions in Section 4.5, we distinguish between the following categories of calls:

- *Fully resolved calls* to `Class.forName` for which all potential targets are class name constants.
- *Partially resolved calls* are calls that have at least one class name string constant propagating to them and are not fully resolved.
- *Unresolved calls* are calls that have no class name string constants propagating to them, only non-constant external sources requiring a specification.

The columns subdivide the total number of calls (T) into fully resolved calls (FR), partially resolved (PR), and unresolved (UR) calls. In the case of Local analysis, there are no partially resolved calls — calls are either fully resolved to constant strings or unresolved. Similarly, in the case of Sound analysis, all calls are either fully resolved or unresolved, as further explained in Section 4.6.5.

4.6.3 Local Analysis for Reflection Resolution (Local)

To provide a baseline for comparison, we implemented a local intra-method analysis that identifies string constants passed to `Class.forName`. This analysis catches only those reflective calls that can be resolved completely within a single method. Because this technique does not use interprocedural points-to results, it cannot be used for identification of specification points. Furthermore, because for method invocations and field accesses the names of the method or field are typically *not* locally defined constants, we do not perform resolution of method calls and field accesses in Local.

A significant percentage of `Class.forName` calls can be fully resolved by local analysis, as demonstrated by the numbers in column 4, Figure 4.18. This is partly due to the fact that it is actually quite common to call `Class.forName` with a constant string parameter for side-effects of the call, because doing so invokes the class constructor. Another common idiom contributing the number of calls resolved by local analysis is `T.class`, which is converted to a call to `Class.forName` and is *always* statically resolved.

4.6.4 Points-to & Reflection Resolution (Points-to)

Points-to information is used to find targets of reflective calls to `Class.forName`, `Class.newInstance`, `Method.invoke`, etc. As can be seen from Figure 4.18, for all of the benchmarks, Points-to information results in more resolved `Class.forName` calls and fewer unresolved ones compared to Local.

Specification Points

Quite frequently, some sort of specification is required for reflective calls to be fully resolved. Points-to information allows us to provide the user with a list of specification points where inputs needs to be specified for a conservative answer to be obtained. Among the specification points we have encountered in our experiments, calls to `System.getProperty` to retrieve a system variable and calls to `BufferedReader.readLine` to read a line from a file are quite common. Below we provide a typical example of providing a specification.

Example 4.9. This example describes resolving reflective targets of a call to `Class.newInstance` in `javax.xml.transform.FactoryFinder` in the JDK in order to illustrate the power and limitation of using points-to information. Class `FactoryFinder` has a method `Class.newInstance` shown in Figure 4.19. The call to `Class.newInstance` occurs on line 9. However, the exact class instantiated at runtime depends on the `className` parameter, which is passed into this function. This function is invoked from a variety of places with the `className` parameter being read from initialization properties files, the console, etc. In only one case, when

```
private static Object newInstance(String className,
        ClassLoader classLoader) throws ConfigurationError {
try {
    Class spiClass;
    if (classLoader == null) {
        spiClass = Class.forName(className);
    }
    ...
    return spiClass.newInstance();
} catch (...)
    ...
}
```

Figure 4.19: Reflection resolution using points-to results in `javax.xml.transform.FactoryFinder` in the JDK.

`Class.newInstance` is called from another function `find` located in another file, is the `className` parameter a string constant.

This example makes the power of using points-to information apparent — the `Class.newInstance` target corresponding to the string constant is often difficult to find by just looking at the code. The relevant string constant was passed down through several levels of method calls located in a different file; it took us more that five minutes of exploration with a powerful code browsing tool to find this case in the source. Resolving this `Class.newInstance` call also requires the user to provide input for four specification points: along with a constant class name, our analysis identifies two specification points, which correspond to file reads, one access of system properties, and another read from a hash table. □

In most cases, the majority of calls to `Class.forName` are fully resolved. However, a small number of unresolved calls are potentially responsible for a large number of specification points the user has to provide. For POINTS-TO, the average number of specification points per invocation site ranges from 3 for `freetts` to 9 for `gruntspud`. However, for `jedit`, the average number of specification points is 422. Specification points computed by the pointer analysis-based approach can be thought of as “hints” to the user as to where provide specification.

In most cases, the user is likely to provide specification at program input points where he knows what the input strings may be. This is because at a reflective call it may be difficult to tell what all the constant class names that flow to it may be, as illustrated by Example 9. Generally, however, the user has a choice. For problematic reflective calls like those in `jedit` that produce a high number of specification points, a better strategy for the user may be to provide reflective specifications at the `Class.forName` *calls themselves* instead of laboriously going through all the specification points.

4.6.5 Casts & Reflection Resolution (Casts)

Type casts often provide a good first static approximation to what objects can be created at a given reflective creation site. There is a significant increase in the number of `Class.forName` calls reported in Figure 4.18 in a few cases, including 93 newly discovered `Class.forName` calls in `gruntspud` that appear due to a bigger call graph when reflective calls are resolved. In all cases, the majority of `Class.forName` calls have their targets at least partially resolved. In fact, as many as 95% of calls are resolved in the case of `jedit`.

As our experience with the Java reflection APIs would suggest, most `Class.newInstance` calls are post-dominated by a cast operation, often located within only a few lines of code of the `Class.newInstance` call. However, in our experiments, we have identified a number of `Class.newInstance` call sites, which were not dominated by a cast of any sort and therefore the return result of `Class.newInstance` could not be constrained in any way. As it turns out, most of these unconstrained `Class.newInstance` call sites are located in the JDK and `sun.*` sources, Apache libraries, etc. Very few were found in application code.

The high number of unresolved calls in the JDK is due to the fact that reflection use in libraries tends to be highly generic and it is common to have "`Class.newInstance` wrappers" — methods that accept a class name as a string and return an object of that class, which is later cast to an appropriate type in the caller method. Since we rely on *intraprocedural* post-dominance, resolving these calls is beyond our scope. However, since such "wrapper" methods are typically called from multiple invocation sites and different sites can resolve to different types, it is unlikely that a precise approximation of the object type returned by `Class.newInstance` is possible in these cases at all.

Precision of Cast Information

Many reflective object creation sites are located in the JDK itself and are present in all applications we have analyzed. For example, method `lookup` in package `sun.nio.cs.AbstractCharsetProvider` reflectively creates a subclass of `Charset` and there are 53 different character sets defined in the system. In this case, the answer is precise because all of these charsets can conceivably be used depending on the application execution environment. In many cases, the cast approach is able to *uniquely* pinpoint the target of `Class.newInstance` calls based on cast information. For example, there is only one subclass of class `sun.awt.shell.ShellFolderManager` available to `gruntspud`, so, in order for the cast to succeed, it must be instantiated.

In general, however, the cast-based approach provides an imprecise upper bound on the call graph that needs to be analyzed. Because the results of `Class.newInstance` are occasionally cast to very wide types, such as `java.lang.Cloneable`, many potential subclasses can be instantiated at the `Class.newInstance` call site. The cast-based approach is likely to yield more precise results on applications that use Java generics, because those applications tend to use more narrow types when performing type casts.

4.6.6 A Sound Call Graph Approximation (Sound)

Providing a specification for unresolved reflective calls allows us to achieve a sound approximation of the call graph. In order to estimate the amount of effort required to come up with a specification for unresolved reflective calls, we decided to start with the Points-to call graph version and then add a reflection specification until all reflective calls become resolved. Because providing a specification allows us to discover more of the call graph, two or three rounds of specification were required as new portions of the program became available. In practice, we would start without a specification and examine all unresolved calls and specification points corresponding to them. Then we would come up with a specification and feed it back to the call graph construction algorithm until the process converges.

Coming up with a specification is a difficult and error-prone task that requires looking at a large amount of source code. It took us about ten hours to incrementally devise an appropriate specification and ensure its completeness by rerunning the call graph construction algorithm to make certain all reflective constructs are fully resolved. After providing a reflection specification stringing with POINTS-TO, we then estimate how much of the user-provided specification can be avoided if we were to rely on type casts instead.

Specification Statistics

The first part of Figure 4.20 summarizes the effort needed to provide specifications to make the call graph sound. The second column shows the number of specifications of the form

```
reflective call site => type
```

as exemplified by Figure 4.15. Columns 3–5 show the number of reflection calls sites covered by each specification, breaking them down into sites that located within library vs. application code. As can be seen from the table, while the number of invocation sites for which specifications are necessary is always around 20, only a few are part of the application. Moreover, in the case of `jfreechart`, *all* of the calls requiring a specification are part of the library code.

Since almost all specification points are located in the JDK and library code, specification can be shared among different applications. Indeed, there are only 40 *unique* invocation sites requiring a specification across all the benchmarks. Column 6 shows the average number of types specified per reflective call site. Numbers in this columns are high because most reflective calls within the JDK can refer to a multitude of implementation classes.

The second part of Figure 4.20 estimates the specification effort required if were were to start with a cast-based call graph construction approach. As can be seen from columns 8–10, the number of `Class.forName` calls that are not constrained by a cast operation is quite small. There are, in fact, only 14 unique invocation sites — or about a third of invocation sites required for Points-to. This suggests that the effort required to provide a specification to make Casts sound is considerably smaller than our original effort that starts with Points-to. However, the cast-based approximation is in many cases quite imprecise, which artificially inflates the call graph size.

Specification Difficulties

In some cases, determining meaningful values to specify for `Class.forName` results is quite difficult, as shown by the example below. One such problematic example was the BeanShell interpreter in `jedit` first described in Section 4.3.7.

Example 4.10. In order to come up with a conservative superset of classes that may be invoked by the BeanShell interpreter for a *given installation* of `jedit`, we parse the scripts that are supplied with `jedit` to determine imported Java classes they have access to. (We should note that this specification is only sound for the default configuration of `jedit`; new classes may need to be added to the specification if new macros become available.)

It took us a little under an hour to develop appropriate Perl scripts to do the parsing of 125 macros supplied with `jedit`. The `Class.forName` call can instantiate a total of 65 different types, which is, of course, an improvement over the overly conservative approximation that assumes that *any* class in the system may be instantiated through the reflective call. □

We should emphasize that the conservativeness of the call graph depends on the conservativeness of the user-provided specification. If the specification missed potential relations, they will be also omitted from the call graph. Furthermore, a specification is typically only conservative for a given configuration of an application: if initialization files are different for a different program installation, the user-provided specification may no longer be conservative.

	Starting with Strings					**Starting with Casts**				
Benchmark	**Specs**	**Sites**	**Libs**	**App**	**Types/site**	**Specs**	**Sites**	**Libs**	**App**	**Types/site**
`jgap`	1,068	25	21	4	42.72	16	2	2	0	8.0
`freetts`	964	16	14	2	60.25	0	4	3	1	0.0
`gruntspud`	1,014	27	26	1	37.56	18	4	4	0	4.5
`jedit`	1,109	21	19	2	52.81	63	3	2	1	21.0
`columba`	1,006	22	21	1	45.73	16	2	2	0	8.0
`jfreechart`	1,342	21	21	0	63.90	18	4	4	0	4.5

Figure 4.20: User-provided specification statistics.

Remaining Unresolved Calls

Somewhat surprisingly, there are *still* some `Class.forName` calls that are not fully resolved given a user-provided specification, as can be seen from the last column in Figure 4.18. In fact, this is not a specification flaw: no valid specification is *possible* for those cases, as explained below.

Example 4.11. The audio API in the JDK includes method `javax.sound.sampled.AudioSystem.getDefaultServices`, which is not called in Java version 1.3 and above. A `Class.forName` call within that method resolves to constant `com.sun.media.sound.DefaultServices`, however, this class is absent in post-1.3 JDKs. However, since this method represents dead code, our answer is still sound. Similarly, other unresolved calls to `Class.forName` located within code that is not executed for the particular application configuration we are analyzing refer to classes specific to MacOS and unavailable on Linux, which is the platform we performed analysis on. In other cases, classes were unavailable for JDK version 1.4.2_08, which is the version we ran our analysis on. □

4.6.7 Effect of Reflection Resolution on Call Graph Size

Figure 4.21 compares the number of classes and methods across different analysis versions. The last column shows by how many times the number of classes or methods in the SOUND version exceeds that for NONE. Local analysis does not have any significant effect on the number of methods or classes in the call graph, even though most of the calls to `Class.forName` can be resolved with local analysis. This is due to the fact that the vast majority of these calls are due to the use of the `T.class` idiom, which typically refer to classes that are already within the call graph. While these trivial calls are easy to resolve, it is the analysis of the other "hard" calls with many potential targets that leads to a substantial increase in the call graph size.

Using POINTS-TO increases the number of classes and methods in the call graph only moderately. The biggest increase in the number of methods occurs for `jedit` (293 methods). Using CASTS leads to significantly bigger call graphs, especially for

Classes						
					Sound	
Benchmark	None	Local	Points-to	Casts	Classes	Sound/None
`jgap`	264	264	268	276	1,569	5.94
`freetts`	309	309	309	351	1,415	4.58
`gruntspud`	1,258	1,258	1,275	2,442	2,784	2.21
`jedit`	1,660	1,661	1,726	2,152	2,754	1.66
`columba`	961	962	966	1,151	2,339	2.43
`jfreechart`	884	881	886	1,560	2,340	2.65

Methods						
					Sound	
Benchmark	None	Local	Points-to	Casts	Classes	Sound/None
`jgap`	1,013	1,014	1,038	1,075	6,676	6.58
`freetts`	1,357	1,358	1,358	1,545	5,499	4.05
`gruntspud`	7,321	7,321	7,448	14,164	14,368	1.96
`jedit`	11,230	11,231	11,523	13,487	16,003	1.43
`columba`	5,636	5,642	5,652	6,199	12,001	2.13
`jfreechart`	5,374	5,374	5,392	8,375	12,111	2.25

Figure 4.21: Number of classes and methods in the call graph for different analysis versions.

`gruntspud`, where the increase in the number of methods compared to None is almost two-fold.

The most noticeable increase in call graph size is observed in version Sound. Compared to None, the average increase in the number of classes is 3.2 times the original and the average increase for the number of methods is 3 times the original. The biggest increase in the number of methods occurs in `gruntspud`, with over 7,000 extra methods added to the graph.

Figure 4.21 also demonstrate that the lines of code metric is not always indicative of the size of the final call graph — programs are listed in the increasing order of line counts, yet, `jedit` and `gruntspud` are clearly the biggest benchmarks if we consider the method count. This can be attributed to the use of large libraries that ship with the application in binary form as well as considering a much larger portion of the JDK in version Sound compared to version None.

4.6.8 Running Times

Figure 4.22 presents the running times for the different versions our our analysis. For each analysis version, we specify the `bddbddb` solver time as well as the total wall clock time, which includes the time to load the relations and save the program representation and call graph information. The overhead involved in these steps can range from 38% to over 100%. When there is a client analysis that uses the results

	None		Local		Points-to		Casts	
Benchmark	**Solver**	**Total**	**Solver**	**Total**	**Solver**	**Total**	**Solver**	**Total**
`jgap`	30	54	34	61	43	73	477	692
`freetts`	16	32	19	35	23	39	180	250
`gruntspud`	193	268	239	318	392	481	5,702	5,860
`jedit`	836	1,236	983	1,394	1,925	2,401	0	7,300
`columba`	106	158	125	179	173	237	1,161	1,456
`jfreechart`	272	395	335	462	450	592	2,578	3,351

Figure 4.22: Running times for different analysis versions, in seconds.

of reflection resolution, these additional saving operations can be avoided by running the client analysis together with reflection resolution while the relevant relations are still within `bddbddb`. While the version with no reflection resolution runs relatively fast, other analysis versions take considerably more time to complete.

4.7 Chapter Summary

This chapter presents the first static analysis for call graph construction in Java that addresses reflective calls. Our algorithm uses the results of a points-to analysis to determine potential reflective call targets. When the calls cannot be fully resolved, user-provided specification is requested. As an alternative to providing specification, type cast information can be used to provide a conservative approximation of reflective call targets.

We applied our static analysis techniques to the task of constructing call graphs for six large Java applications, some consisting of more than 190,000 lines of code. Our evaluation showed that as many as 95% of reflective `Class.forName` could at least partially be resolved to statically determined targets with the help of points-to results and cast information *without* providing any specification.

While most reflective calls are relatively easy to resolve statically, *precisely* interpreting some reflective calls requires a user-provided specification. Our pointer analysis-based approach also identified specification points — places in the program corresponding to file and system property read operations, etc., where user input is needed in order to obtain a full call graph. Our evaluation showed that the construction of a specification that makes the call graph conservative is a time-consuming and error-prone task. Fortunately, our cast-based approach can drastically reduce the specification burden placed on the user by providing a conservative, albeit potentially imprecise approximation of reflective targets.

Our experiments confirmed that ignoring reflection results in missing significant portions of the call graph, which is not something that effective static analysis tools can afford. While the local and points-to analysis techniques resulted in only a moderate increase in call graph size, using the cast-based approach resulted in call graphs with as many as 1.5 times more methods than the original call graph. Furthermore, providing a specification resulted in much larger conservative call graphs that were almost 7 times bigger than the original. For instance, in one our benchmark, an additional 7,047 methods were discovered in the conservative call graph version that were not present in the original.

Chapter 5

Runtime Analysis

In this chapter we describe the runtime analysis part of the Griffin project. Runtime analysis results are summarized in Chapter 6.

5.1 Advantages of the Runtime Approach

As part of the Griffin project, we have developed a runtime security protection and recovery system for Web applications. Commonly used dynamic techniques such as application firewalls [152] that rely on pattern-matching and monitor traffic flowing in and out of the application are often a poor solution for SQL injection or cross-site scripting attacks. Such techniques suffer from both false positives and false negatives.

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

Deployment-time security. Runtime analysis can be integrated with the server so that whenever a new Web application is added, it is instrumented automatically.

This removes the risk associated with deploying "unfamiliar", potentially unsafe Web applications. This approach eliminates the "vulnerability window" that stems from the code changing without the static analysis tool being immediately rerun. Moreover, recovery from vulnerabilities can be provided by applying user-provided sanitization.

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

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

Avoids static analysis challenges. Finally, as described in Chapter 3, analyzing Web applications statically can be challenging because of the difficulty of call graph construction and reflection. Runtime analysis avoids these challenges altogether.

The rest of the chapter is organized as follows. Section 5.2 describes the process of translating a PQL query into nondeterministic finite automata (NFAs). Section 5.3 describes the NFAs corresponding to tainted object propagation queries in Chapter 2. Section 5.4 discusses the issue of runtime overhead. Finally, Section 5.5 addresses runtime recovery from exploits.

5.2 Matching PQL Queries at Runtime

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

Whenever the NFA corresponding to the `main` query enters an accept state, one of several outcomes can occur. If the **replaces** clause is present, another event is substituted in place of the event being replaced. This is especially useful for recovery, so that a safe action replaces a potentially unsafe one, as described in Section 5.5. If the **executes** clause is present, the code within the clause will be executed, which is useful for reporting vulnerabilities or terminating the application.

Finding dynamic matches to PQL queries involves the following steps:

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

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

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

Each of these steps is described in detail in Sections 5.2.1 — 5.2.3.

5.2.1 Translation From Queries To State Machines

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

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

A *partial query match* is given by a current state and a set of *bindings* — mappings from variables in a PQL query to objects in the heap at runtime. A state transition specifies the event for which a current state and current bindings transition to the next state and a new set of bindings. Because the same event may be interpreted in different ways by different transitions, a state machine may non-deterministically transition to different states given the same input.

Special Transitions

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

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

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

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

We preprocess the original PQL queries to ease the translation process. No subquery may, directly or indirectly, invoke itself without any intervening events. So, first we eliminate such situations, a process analogous to the elimination of left-recursion from a context-free grammar [2]. Second, excluded events are propagated forward through subquery calls and returns so that each set of excluded events is either at the end of `main` or immediately before a primitive statement.

Transitions Corresponding to Primitive Statements

We now present a syntax-directed approach to constructing the state machine for a query. The reader is encouraged to refer to the PQL grammar in Figure 2.4 as we describe how different primitive statements are translated. Before we can proceed, some additional notation is required.

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

Definition 5.2.1 An attribute in event e with value x is *unifiable* with query statement s and the current set of bindings b if

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

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

Array and field operations. These are the primitive statements that correspond to single events during the execution. For a primitive statement s of type t, the transition from *bef*(s) to *aft*(s) is predicated by getting an input event e also of type t and that the attributes in e must be unifiable with those in statement s and the current bindings. If the attribute refers to an unbound variable v, the pair (v, x) is added to the set of known bindings.

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

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

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

Method invocation and creation points. If s is a method invocation statement, we must match the call and return events for that method, as well as all events between them. To do this, we create a fresh state t and a new event variable v. We create a transition from $\mathit{bef}(s)$ to t that matches the call, and bind v to the ID of the event. We create another transition from t to $\mathit{aft}(s)$ that matches a return with ID v. The skip transition from t back to itself is modified to exclude the match of the return event. Calls and returns are unified in a manner analogous to array and field operations. Object creation is handled in Java by invoking the method "`<init>`", and is translated into NFAs like any other method invocation.

Unification statements. A unification statement denoted by *unifyStmt* in Figure 2.4 is represented by a predicated ϵ transition that requires that the two variables on the left and right have the same value. If one is unbound, it will acquire the value of the other.

5.2.2 Instrumenting the Program

The system instruments all instructions in the target application that match any primitive event or any exclusion event in the query. At an instrumentation point, the pending event and all relevant objects are sent to the query matcher. The matcher updates the state of all pending matches and then returns control to the application. For instance, the NFA that corresponds to a PQL query that concerns calls to method `StringBuffer.toString`() will be notified each time this method is invoked. Moreover, the value of the `this` parameter will be passed to the NFA also.

The matcher does not interfere with the behavior of the application except via completed matches. Therefore, any instrumentation point that can be statically proven to not contribute to any match need not be instrumented.

5.2.3 The Runtime Query Matcher

The matcher begins with a single partial match at the beginning of the `main` query, with no values for any variables. It receives events from the instrumented application and updates all currently active partial matches. For each partial match, each transition from its current state that can unify with the currently processed event produces a new possible partial match where that transition is taken.

Handling Non-Determinism

A single event may be unifiable with multiple transitions from a state, so multiple new partial matches are possible. If a skip transition is present and its predicates pass, the match will persist unchanged. If the skip transition is present but a predicate fails the match transitions to the fail state. If the skip transition is present but a predicate's value is unknown because the variables it refers to as are of yet unbound, then the variable is bound to a value representing "any object that does not violate the predicate." Predicates accumulate if two such objects are unified; unification with any object that satisfies all such predicates replaces the predicates with that object. If the new state has ϵ transitions, they are processed immediately.

Handling Subqueries

If a transition representing a subquery call is available from the new state, a new partial match based on the subquery's state machine is generated. This partial match begins in the subquery's start state and has initial bindings corresponding to the arguments the subquery was invoked with.

A unique subquery ID is generated for the subquery call and associated with the subquery caller's partial match, with the subquery callee's partial match, and with any partial match that results from taking transitions within the subquery callee.

Handling Accept States

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

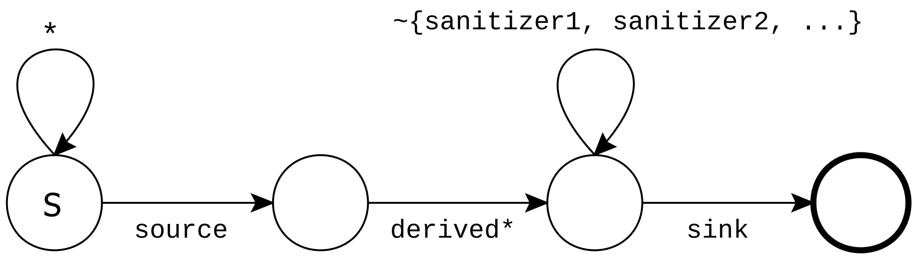


Figure 5.1: State machine that corresponds to the `main` PQL query.

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

5.3 Translating Vulnerability Queries

The previous section presented a generic procedure for translating from PQL queries to NFAs. This section discusses the state machines that are created for the specific vulnerability queries shown in Figures 2.5 — 2.7. For all the NFAs discussed in this section, S marks the start state and thick-edged graph nodes are accept states. For edges, $*$ marks an edge that can be taken on any input. Exclusion notation $\sim \mathtt{e}_1, \mathtt{e}_2, \ldots$ on graph edges marks an edge that can be taken on any input events *except* $e_1, e_2, \ldots$.

Query `main`. The NFA in Figure 5.1 for the `main` PQL query consists of invocations of subqueries `source`, `sink`, and `derived`$*$. This corresponds to a piece of data that is read from a source, derived from using zero or more steps, and then falls into a sink. This exactly matches the notion of a tainted object propagation problem in Section 2.1.1.

It is important to point out that the transition on the `sink` edge leading to the accepting node is only allowed when no sanitizer calls are encountered (sanitizers are denoted by `sanitizer1`, `sanitizer2`, etc.). This is important since it is

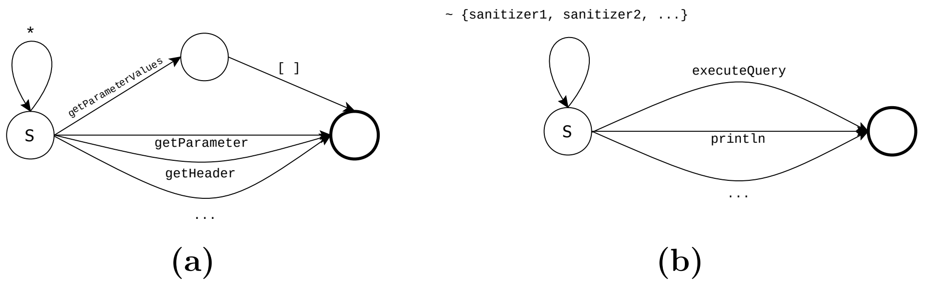


Figure 5.2: State machines corresponding to the (a) `source` and (b) `sink` PQL queries.

possible for `derived`∗ query to complete without encountering a sanitizer. Once the `derived`∗ step finishes, a sanitizer could be applied to the same object as the one passed into a sink.

Query `source`. The `source` NFA shown in Figure 5.2(a) accepts on methods calls to source methods such as `getParameter`, etc. One complication is the treatment of return values of a call to `getParameterValues`. It is required that the returned array be indexed, as represented by the edge marked with “[ ]” for the state machine to accept. A similar technique is used to make values of a map returned from `getParameterMap` tainted, except that several possibilities exist: method `get` needs to be called on the map returned from the call; alternatively, an iterator could be constructed over the map values by calling `values`().`iterator`() and then method `next`() could be called on the iterator.

Queries `sink` and `derived`. Queries `sink` and `derived` consist of an alternation of methods that correspond to sink and derivation descriptors, respectively. Notice that the `sink` and `derived` NFAs in shown in Figures 5.2(b) and 5.3(a) only accepts if no sanitizer is encountered.

Query `derived`∗. The NFA in Figure 5.3(b) is self-recursive and corresponds to zero or more invocations of subquery `derived`. When the `temp` node is reached, a new state machine is created to interpret the recursive invocation of `derived`∗. Eventually, the top branch from the start node will be taken, thus completing the subquery match.

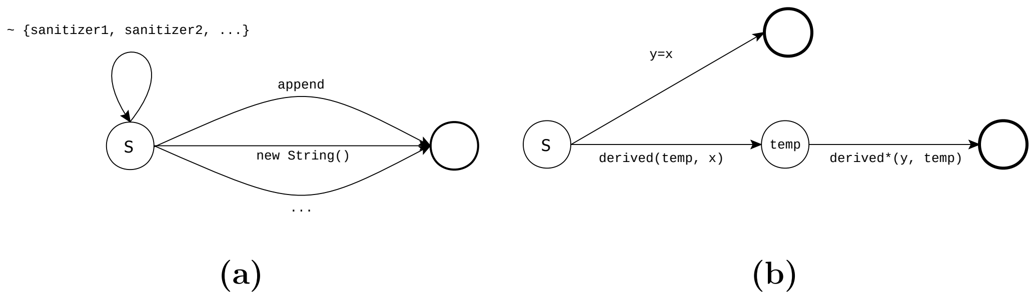


Figure 5.3: State machines corresponding to the (a) `derived` and (b) `derived*` PQL queries.

5.4 Reducing Instrumentation Overhead

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

In order to reduce the overhead further, we use the results of our static analysis, described in Section 3.5.4, to reduce the instrumentation by excluding statements that cannot refer to objects involved in any match of the query. For queries capturing the tainted object problem, we only need to instrument calls on a path from a source to a sink, which account for a small portion of all string-related method calls. Also, as described in Martin et al. [140], instead of collecting full execution traces and post-processing them, our system tracks all the partial matches as the program executes and takes action immediately upon recognizing a match.

5.5 Dynamic Recovery from Vulnerabilities

Figure 5.4 presents an augmented version of query `main` that has recovery capabilities. As can be seen from the augmented query, each operation that can unsafely use tainted data receives a **replaces** clause in the augmented `main` query.

When a possibly relevant sink is reached, any match that has completed and

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

Figure 5.4: Augmented `main` query for recovering from exploits at runtime.

which is consistent with the event being replaced is gathered. If the **replaces** clause is present, the replacing method is executed instead. Since every argument to the **replaces** clause except `sourceObj` appears in the replaced event, `sourceObj` is the only variable that may have multiple values. The replacement method provides a safe alternative for each of the sinks in the query. In general, the replacement method sanitizes tainted values. The kind of sanitization applied is different depending on both the type of vulnerability and the method being replaced.

5.5.1 Built-in Sanitization

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

Example 5.1. SQL sanitization methods `SafePrepare` and `SafeExecute` work by finding all substrings within string `sinkObj` that match any of the possible values for string `sourceObj`. A new SQL query string is constructed with all SQL metacharacters in any such substring quoted. This new query is then passed to methods `prepareStatement` or `executeQuery` invoked on the underlying object, respectively. This is not dissimilar to to the approach taken in Buehrer et al. [26].

To understand how this work for protecting against SQL injections, consider a `sourceObj` that refers to string ′`O′Brian`′. Suppose `sinkObj` refers to string

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

The result of applying `SafePrepare` will be

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

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

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

5.5.2 Shortcomings of Built-in Sanitizers

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

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

will be converted by `hibernate` into

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

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

This illustrates a more general point about applying sanitization: where it needs to be placed is often open for discussion. While our approach of applying it right before the sink works in most cases, it is not necessarily most efficient. In many cases, the proper place to insert sanitization — both in the code and at runtime — is between abstraction boundaries or before a piece of data is placed into a data structure, etc. An example that further illustrates this dilemma is discussed in Section 6.3.4.

Chapter 6

Experimental Results

This chapter summarizes experimental results for the static and runtime analyses described in Chapters 3 and 5. Our analyses are applied to two suite of benchmark applications: large, real-life open source Java programs and a suite of small artificial test cases. Our focus is to find out what kind of vulnerabilities are discovered by static analysis and to determine the effect of static analysis features on the number of false positives. For runtime analysis, our primary focus is to measure the runtime overhead incurred with our dynamic analysis.

6.1 Experimental Setup

In this section we describe the benchmarks we used in our experiments. There are two sets of benchmarks we have used. The first one — Stanford SecuriBench — consists of large real-life open-source applications, most of which are available on Sourceforge [125]; these are the "macro" benchmarks. The second benchmark suite is Stanford SecuriBench Micro [127], a collection of small artificial benchmarks we developed to tests various aspects of static analysis. Both suites of benchmarks are described in more detail below. Both benchmark suites are made publicly available from the Griffin project Web page [126] to foster an exchange of ideas as well as further research in the area of Web application security.

Our static analysis framework was implemented on top of the Joeq compiler research infrastructure [202] and the bddbddb Datalog solver [203]. Runtime instrumentation was performed with the help of the PQL system [139] build on top of Apache BCEL instrumentation framework [57]. Finally, analysis times shown in Figure 6.3 and mentioned elsewhere are obtained on an Opteron 150 machine with 4 GB of memory running Linux.

6.1.1 SecuriBench 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 restrict our attention to artificial micro-benchmarks or student projects that lack the complexities inherent in real applications.

Evaluation Strategy

While some attempts have been made at constructing artificial large-scale Web application benchmarks [58, 164], we believe that real-life programs are much better suited for comparing security tools. We focused on a set of large, representative open-source Web-based J2EE applications, most of which are available on SourceForge. We released the Stanford SecuriBench suite of Web application benchmarks in 2005. We are making these benchmarks publicly available in hopes of fostering collaboration between researchers. So far, SecuriBench consists of 11 real-life open-source Web-based applications written in Java and developed using the J2EE framework.

Benchmark	Version	LOC	Expanded LOC	Files	Classes	Jars
`jboard`	0.3	17,542	95,845	90	311	29
`webgoat`	0.9	17,678	117,207	73	296	27
`jgossip`	1.1	72,795	170,893	537	575	63
`personalblog`	1.2.6	5,635	226,931	38	754	65
`blojsom`	2.30	53,309	332,902	245	395	41
`road2hibernate`	1.1	2,601	352,737	21	796	34
`snipsnap`	1.0-BETA-1	48,220	445,461	446	859	68
`pebble`	1.6-beta1	42,920	449,395	333	945	47
`jorganizer`	0.0.22	14,495	454,735	107	920	35
`roller`	0.9.9	52,089	557,592	276	972	133
Total		**327,284**	**3,203,698**	**2,166**	**6,823**	**542**

Figure 6.1: Summary of information about SecuriBench benchmarks. Applications are sorted by the expanded line of code count, shown in column 4.

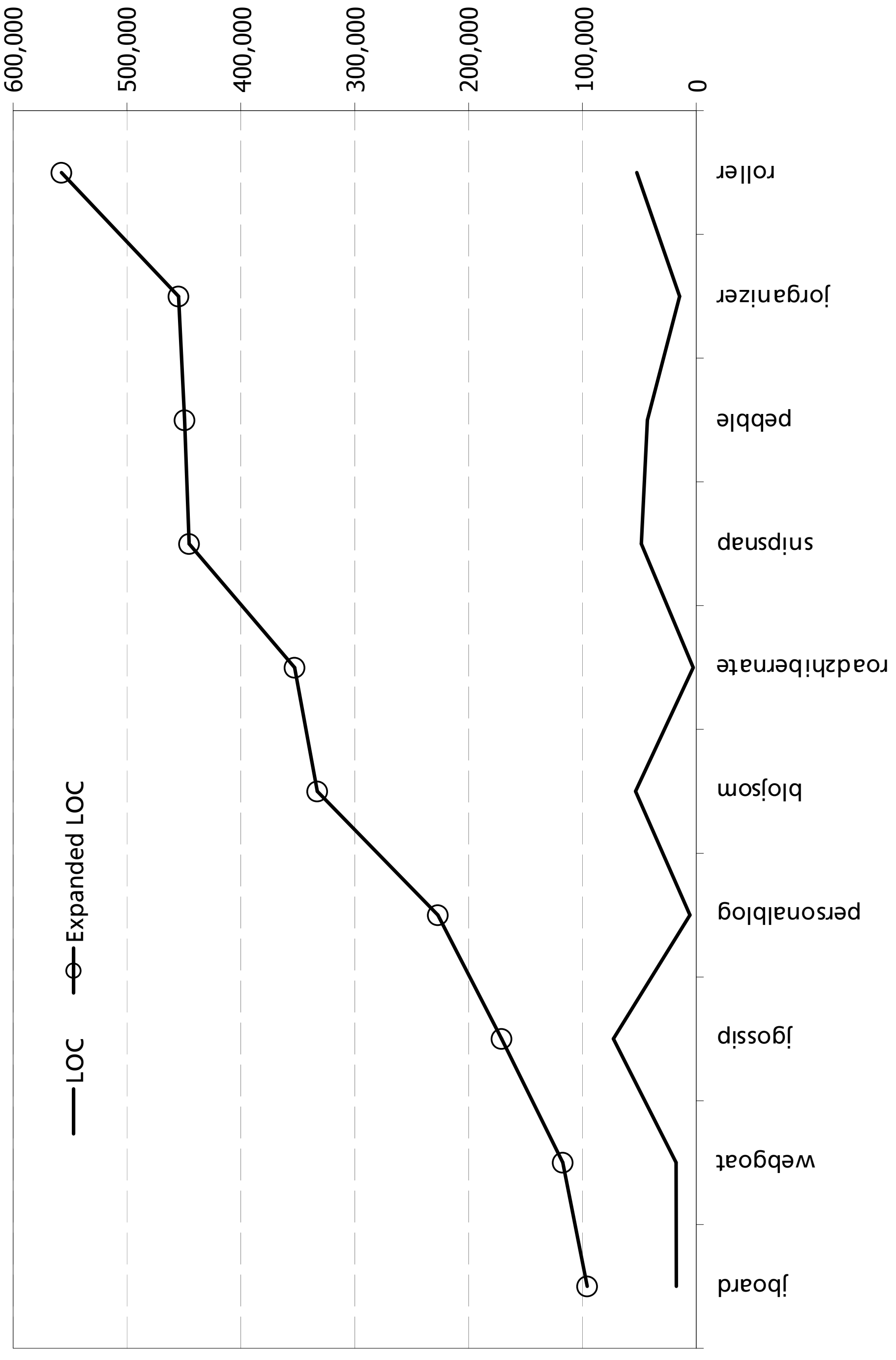


Figure 6.2: Sizes of SecuriBench benchmark applications.

Description of Benchmark Applications

The benchmark applications we used are briefly described below. Applications `jboard`, `jgossip`, `blojsom`, `personalblog`, `snipsnap`, `pebble`, and `roller` are Web-based bulletin board and blogging applications. `jorganizer` is an online personal information manager. `webgoat` is a J2EE application designed by the Open Web Application Security Project [157, 158] 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 applications that we developed for exploring injection vectors into the `hibernate` library.

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 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.

Application Statistics

Figure 6.1 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 amount of code that needs to be analyzed 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 of application size called "expanded lines of code" is given in the fourth column of Figure 6.1. It combines the total number of lines within the application and library code. To compute expanded line of code statistics we created a tool that aggregated lines of code for all classes that are present in the call graph. When the source code was unavailable for library classes, we used jad, a popular Java disassembler [110] to obtain a line count estimate. Figure 6.2 contains a graphical summary of the application size data. Columns 5 — 7 summarize the numbers of files, classes, and jar (or Java ARchive) libraries each application contains. The high number of jar files shown in the last column can be explained by the fact that many applications rely heavily on external libraries, which are distributed as jars.

Analysis Times

Figure 6.3 summarizes the running times for various phases of the static analysis for SecuriBench applications. The second column of the table shows the times to pre-process the application to create relations accepted by the pointer analysis. It also includes the time involved in call graph construction as well as inlining the call graph to achieve better object naming, as described in Section 3.4.4. Call graph inlining is a major contributing factor to the cost of call graph construction as well as program relation generation. Call graph inlining also makes the amount of code that needs to be analyzed quite a bit larger, slowing down further analysis phases.

The third column of the table shows the time taken by a call graph numbering pass, required to perform a context-sensitive pointer analysis described in [205]. The fourth column shows context-sensitive points-to analysis times. Finally, columns 5–7 show the times to find the vulnerabilities statically, as described in Section 3.5.1. These times are broken down into the times it takes to find sources, sinks, and to solve the tainted object propagation problem.

6.1.2 SecuriBench Micro Benchmarks

The second testbed for our experiments is a suite of more than a hundred small synthetic benchmarks designed to test various aspects of static analysis coverage and precision. All SecuriBench Micro benchmarks are small J2EE servlets that can

Benchmark	**Call graph**		**Points-to**	**Taint analysis**			**Total**
	Construction	**Numbering**	**analysis**	**Source**	**Sink**	**Taint**	
`jboard`	192	23	14	7	5	12	253
`webgoat`	667	135	154	30	30	222	1,238
`jgossip`	649	99	55	19	15	48	885
`personalblog`	1,901	286	367	58	52	40	2,704
`blojsom`	392	62	43	15	14	46	572
`road2hibernate`	2,963	368	321	55	44	461	4,212
`snipsnap`	1,417	191	156	38	28	186	2,016
`pebble`	1,168	161	103	30	25	120	1,607
`jorganizer`	2,811	246	69	60	61	60	3,307
`roller`	3,414	481	97	87	70	335	4,484

Figure 6.3: Times, in seconds, to perform preprocessing, points-to, and taint analysis.

Test category	**Category description**	**Tests**
basic	Handling of various simple cases	41
collections	Handling of standard collections	14
interprocedural	Handling of flow across methods	14
arrays	Handling of flow array element accesses	9
predicates	Handling of conditionals	9
sanitizers	Handling of standard library sanitization routines	6
aliasing	Handling of pointer aliasing	6
data structures	Handling of recursive data structures	6
strong updates	Handling of strong updates to variables	5
factories	Handling of factory methods	3
session handline	Handling of objects placed inside the `Session` structure	3
Total		**116**

Figure 6.4: SecuriBench Micro test case categories.

be deployed on a standard application server. SecuriBench Micro benchmarks were developed by us as we worked on the static analysis portion of the analysis and are made available to the public [127]. To date, some researchers and tool vendors have decided to use SecuriBench Micro. We hope that others will find this benchmark suite in their own research.

SecuriBench Micro benchmarks are subdivided into 11 categories, each designed to exercise a different portion of the analysis. More information about benchmark categories is given in Figure 6.4. Static analysis results are conservative, meaning that all known vulnerabilities in SecuriBench Micro benchmarks are found by static analysis. Comparing against a known set of vulnerabilities gives us additional assurance of the correctness of our implementation.

Example 6.1. A sample test case from SecuriBench Micro is shown in Figure 6.5. This case tests `String` and `StringBuffer` handling within the static analyzer. Class `Basic10` is a simple J2EE servlet that implements method `doGet`, which is invoked when a Web form is submitted. Objects `req` and `resp` give access to servlet input and output, respectively. In this case, a cross-site scripting vulnerability is enabled

```
 1.     public class Basic10 extends BasicTestCase implements MicroTestCase {
 2.         protected void doGet(HttpServletRequest req, HttpServletResponse resp)
 3.             throws IOException
 4.         {
 5.             String s1 = req.getParameter("name");       /* TAINTED */
 6.             String s2 = s1;
 7.             String s3 = s2;
 8.             String s4 = s3;
 9.             StringBuffer b1 = new StringBuffer(s4);
10.             StringBuffer b3 = b1;
11.             String s5 = b3.toString();
12.             String s6 = s5;
13.
14.             PrintWriter writer = resp.getWriter();
15.             writer.println(s6);                         /* BAD */
16.         }
17.     }
```

Figure 6.5: Sample benchmark from the SecuriBench Micro suite.

via a call to `println` on line 15. The source is a tainted parameter `s1` obtained on line 5. Derived objects that participate in the vulnerability trace are `s2`, `s3`, `s4`, `b1`, `b3`, `s5`, and `s6`. □

We do not report the runtime for the micro benchmarks separately, as the variation among them is small. The total time to run the analysis on a single benchmark is a little over 3 minutes. The preprocessing time is about 2.5 minutes for relation generation. This is not surprising, as even this small program consists a total of 160 classes and 799 methods when all the libraries are counted. The context-sensitive points-to analysis takes 12 seconds and the total time taken by the three stages of the taint analysis is 25 seconds.

SecuriBench Micro Vulnerabilities

Figure 6.1.2 summarizes the vulnerabilities found in SecuriBench Micro. For each benchmark category shown in column 1, column 2 shows the number of test cases within that category, column 3 shows the number of vulnerabilities discovered by our static analysis, and column 4 shows the number of false positives. The results provided by the Griffin static analysis are conservative, i.e. the analysis finds all vulnerabilities

```
 1. public class Arrays8 extends BasicTestCase implements MicroTestCase {
 2.     protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
 3.         String name = req.getParameter("name");     /* TAINTED */
 4.         String[] array = new String[] {name, "abc"};
 5.
 6.         PrintWriter writer = resp.getWriter();
 7.         writer.println(array[0]);                   /* BAD */
 8.         writer.println(array[1]);                   /* OK */
 9.     }
10. }
```

Figure 6.6: A false positive related to array handling.

contained in the test cases. The total number of vulnerabilities exceeds the number of tests because some tests have more than one vulnerability embedded in them. The analysis reports false positives in a number of test cases; the overall false positive rate, however, is relatively low: 16% for SecuriBench Micro. Many false positives are reported in "arrays" and "predicates" categories, as illustrated below.

Example 6.2. An example of such a false positive in the "arrays" category of SecuriBench Micro is presented in Figure 6.6. While the call to `println` on line 7 corresponds to a legitimate vulnerability because a tainted piece of data obtained on line 3 propagates to the output, the call on line 8 is benign because `array[1]` is the string constant `"abc"`. The false positive is due to the fact that our static analysis has the same representation for all elements of the `String` array `array`. □

This is not surprising, as our static technique performs no array index disambiguation; as a result, all array elements have the same taint status. Nor does our technique pay attention to predicates so execution paths that are infeasible at runtime are considered possible by the static analysis. However, neither of these analysis features appear to be important for macro benchmarks.

6.2 Static Analysis Results

In this section we summarize the experiments we performed and described the security violations we found. We start by describing our experimental setup, describe some

Test category	**Tests**	**Vulnerabilities**	**False positives**
basic	41	59	3
collections	14	13	3
interprocedural	14	17	4
arrays	9	8	4
predicates	9	4	4
sanitizers	6	4	2
aliasing	6	11	1
data structures	6	5	1
strong updates	5	1	2
factories	3	3	0
session handling	3	3	1
Total	**116**	**128**	**25**

Figure 6.7: SecuriBench Micro static analysis result summary.

representative vulnerabilities found by our analysis, and analyze the impact of analysis features on precision.

6.2.1 Experimental Setup

The implementation of our static analysis system is based on the `joeq` Java compiler and analysis framework [202]. We applied static analysis to look for all tainted object propagation problems described in Chapter 2. We used a total of 28 source, 18 sink, and 29 derivation descriptors in our experiments. Since most sanitizers manufacture and return a new, fresh object, it was not necessary to mention most. However, we added two sanitizers, which might under certain conditions return the original string that is passed in.

6.2.2 Summary of Discovered Vulnerabilities

This presents statistics about the vulnerabilities discovered by the static analysis in both benchmark suites.

Benchmark	**Sources**	**Sinks**	**Vulnerabilities**	**False positives**
`jboard`	1	8	2	0
`blueblog`	11	32	2	0
`webgoat`	13	47	5	0
`personalblog`	45	22	40	0
`blojsom`	34	29	6	0
`snipsnap`	138	88	16	0
`road2hibernate`	2	13	13	0
`pebble`	123	39	0	0
`roller`	41	66	5	147
`jorganizer`	116	20	7	0
`jgossip`	22	29	0	0
Totals			**98**	**147**

Figure 6.8: SecuriBench static analysis result summary.

SecuriBench Vulnerabilities

The static analysis described in this report reports a total of 245 potential security violations in our 11 benchmarks, out of which 98 turn out to be security errors, while 147 are false positives. *All but two* of SecuriBench benchmarks had at least one security vulnerability. Moreover, except for errors in `webgoat` and one HTTP splitting vulnerability in `snipsnap` [63], none of these security errors had been reported before. Furthermore, *only one* benchmark in our experiments produced false positives, which demonstrates the precision of our technique.

It is not uncommon to have *multiple paths* connecting the same source and sink pair. To avoid this sort of double-counting in the data we present below, we have decided to use the number of source-sink pairs when reporting the numbers of vulnerabilities and false positives.

6.2.3 Validating the Vulnerabilities

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

	SQL injections	**HTTP splitting**	**Cross-site scripting**	**Path traversal**	**Total**
Header manipulation	0	1	0	0	**1**
Parameter manipulation	55	11	2	10	**78**
Cookie poisoning	0	1	0	0	**1**
Non-Web inputs	15	0	0	3	**18**
Total	**70**	**13**	**2**	**13**	**98**

Figure 6.9: Classification of the vulnerabilities found. Each cell in the table corresponds to a combination of a source type (shown in rows) and sink type (shown in columns).

dynamically. It may not be possible to construct meaningful malicious input. Exploits may also be ruled out because of the particular configuration of the application.

However, 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. Code that is dead at one point in the application lifetime may become not dead after a small code change is introduced. 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.

Reporting Vulnerabilities

After we ran our analysis, we manually examined all the errors reported to make sure they represent security errors. Since our knowledge of the macro 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. As mentioned in Section 3.8, the fact that vulnerability traces consist of objects makes them sometimes more difficult to interpret. Constructing precise exploit scenarios may be difficult because multiple program variables refer may to the same object.

Our approach was to report vulnerability warnings as exploitable unless the code they were contained in was obviously unreachable. While somewhat unlikely, it is possible that some of the reported errors were in dead code, but the fact that application maintainers were willing to fix them enhanced our believe in this approach. In some cases where the control flow was particularly complex, we had to run the application in the debugger to see if some of the errors were indeed exploitable. This was the case with the false positives in `roller`: we were not able to ascertain that they corresponded to realizable control flow paths and therefore these warnings were classified as false positives.

```
public static String filterNewlines(String s) {
    if (s == null) {
        return null;
    }

    StringBuffer buf = new StringBuffer(s.length());

    // loop through every character and replace if necessary
    int length = s.length();
    for (int i = 0; i < length; i++) {
        switch (s.charAt(i)) {
            case '\n':
            break;
            default :
            buf.append(s.charAt(i));
        }
    }

    return buf.toString();
}
```

Figure 6.10: Typical string sanitization routine in `pebble`.

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.

Control Flow

It is important to remind the reader that our current analysis ignores control flow. Without analyzing predicates that affect the flow of control, 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.

Most large Web-based application perform some form of input checking. However, as in the case of the vulnerabilities we found, it is common that some of the checks are missed. Our analysis did not produce any false positives that were due to the lack of predicate analysis. We believe that this is largely because sanitization is primarily done by manufacturing a new, clean version of the original string, as exemplified by the sanitizer in Figure 6.10. Method `filterNewlines` performs character-by-character blacklisting of the input. If the character `s.charAt(i)` being processed is outside the black list, which consists of the new line character, it is added to the output buffer. Sanitization in scripting languages such as PHP is usually done by applying boolean tests to potentially malicious input and therefore requires path sensitivity [88, 95, 207].

6.2.4 Classification of Errors

This section presents a classification of all the errors we found in the SecuriBench suite on 11 large applications.

Vulnerability Summary

A summary of our experimental results is presented in Figure 6.2. Columns 2 and 3 list the number of source and sink objects for each benchmark. It should be noted that the numbers of sources and sinks for all of these applications are 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 and the number of false positives in columns 4 and 5.

Vulnerability Classification

Figure 6.9 presents a classification of the 98 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 15 potential SQL injection attacks caused by non-Web sources. Notice that we chose not to distinguish between parameter manipulation and hidden field manipulation when presenting the results, because these vulnerabilities manifest themselves similarly at the source level.

Overall, parameter manipulation was the most common technique to inject malicious data (78 cases) and SQL injection was the most popular exploitation technique (70 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"));
```

Attack Vectors in Library Code

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 several attack vectors in libraries, including a couple in a commonly used Java library `hibernate`, another in `castor` database used in `jorganizer`, and another in the J2EE implementation itself. Some of these attack vectors are described below.

6.3 Discussion of Static Analysis Results

This section describes some of the vulnerabilities we found in more detail. It also presents several studies we have performed to further explore and characterize the effectiveness of various static analysis features.

6.3.1 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. Several programs in our benchmark suite, including `personalblog`, `snipsnap`, and `roller` 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 lead to vulnerabilities, as exemplified by two errors we found in `personalblog`.

A few other public methods in the `hibernate` APIs, such as `iterate` and `delete` also turn out to be attack vectors as well, even though these API methods are not used in client code of our benchmarks. We validated these vulnerabilities by adding them as test cases to `road2hibernate` code and executing it under the debugger[1].

The situation with `hibernate` APIs 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 common libraries, etc. in order to protect their clients.

6.3.2 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 that is specified as part of the HTTP protocol. In the HTTP protocol, method `TRACE` is used to echo the contents of an HTTP request back to the client for debugging purposes [52].

However, when a `TRACE` request is issued the contents of user-provided headers are sent back verbatim to the browser, thus enabling cross-site scripting attacks. The vulnerable code extracted from class `javax.servlet.http.HttpServlet`, JDK version 1.4.1 is shown in Figure 6.11. On lines 13–17, user-controlled tainted header values are appended to `responseString`. This string is subsequently sent back to the user's browser on line 26.

In fact, this variation of cross-site scripting caused by a vulnerability in HTTP

[1]In a recently released version 3 of the `hibernate` library, a new SQL injection-prone API was introduced, which can now be used instead of the unsafe one. The user is now required to create a parse tree for the SQL expression they want to send to the database by hand.

```
    protected void doTrace(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException
        {

            int responseLength;

            String CRLF = "\r\n";
            String responseString = "TRACE "+ req.getRequestURI()+
                " " + req.getProtocol();

            Enumeration reqHeaderEnum = req.getHeaderNames();

            while( reqHeaderEnum.hasMoreElements() ) {
                String headerName = (String)reqHeaderEnum.nextElement();
                responseString += CRLF + headerName + ": " +
                    req.getHeader(headerName);
            }

            responseString += CRLF;

            responseLength = responseString.length();

            resp.setContentType("message/http");
            resp.setContentLength(responseLength);
            ServletOutputStream out = resp.getOutputStream();
            out.print(responseString);
            out.close();
            return;
        }
```

Figure 6.11: Cross-site scripting vulnerability in method `doTrace`.

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, both of which avoid the possibility of cross-site tracing altogether [69]. Since the vulnerable code is part of the JDK and is present in many Java applications, we are only counting this vulnerability once.

Analysis version	**False positives**
Context-insensitive	114
Context-sensitive	84
Context-sensitive and better object naming	43
Context- and map-sensitive, better object naming	5
Context- and map-sensitive, better object naming, sanitizers added	0

Figure 6.12: Number of false positives in `blojsom` for different pointer analysis versions.

6.3.3 Effect of Analysis Features on False Positives

Figure 6.12 shows the effect of analysis features on reducing the number false positives in `blojsom`, one of the medium-sized benchmarks in SecuriBench. In addition to the false positives studied in this section, our analysis discovered 6 real vulnerabilities in `blojsom` code.

The initial "base" analysis version that uses a plain context-insensitive pointer analysis with no additional enhancements produces a large number of false positives. Adding analysis features one by one reduces their number significantly. One distinguishing characteristic of `blojsom` is that map sensitivity plays an important role. This is not the case for some of the other benchmarks. The last row of the table corresponds to the case where we add missing sanitizers to the specification, thus suppressing five more warnings.

6.3.4 Connectivity Between Sources and Sinks

It is not uncommon for a single "unprotected" source to enable many vulnerabilities. Similarly, an sink that does not have sanitization preceding it can also create many vulnerabilities. We have observed this pattern in several applications and believe it to be quite widespread. Knowing how sources and sinks are connected within an application can lead to a better placement of sanitizers.

Example 6.3. Figure 6.13 presents an example of source-sink connectivity extracted from `personalblog`. A node of the graph is a static object approximation. Edges

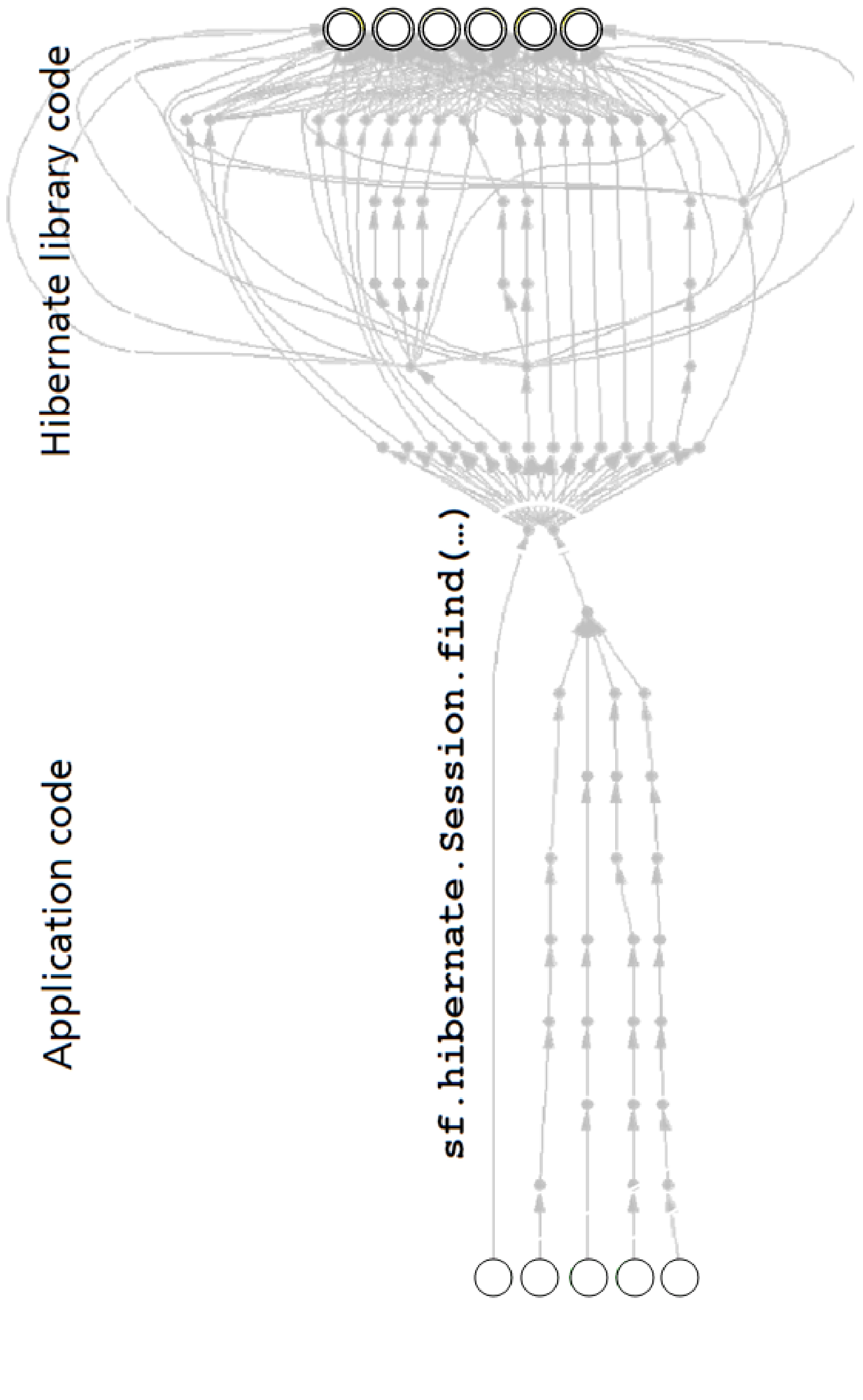


Figure 6.13: Source-sink connectivity in `personalblog`. Each node of the graph represents a static representation of an object, i.e. an allocation site in the program. Single- and double-bordered circles represent source and sink objects, respectively.

correspond to derivation edges. Sources and sinks are represented by large single- and double-bordered circles, respectively. Notice that this graph represents a small portion of the total object connectivity graph. In this case, we focus on objects that are reachable from the sources that are on at least one path leading to a sink. Notice that the bipartite source-sink graph shown in Figure 6.13 is fully connected, i.e. every source can reach every sink.

The graph in Figure 6.13 can be naturally subdivided into two parts. The first portion on the graph on the left corresponds to the taint propagation that happens in the `personalblog` code. The second portion on the right represents taint propagation through the `hibernate` object persistence library. Notice that the `hibernate` portion of the graph is much more tightly connected. This is because string parsing and processing code in `hibernate` contains many `String` constructors and there are many tainted `String`s flowing to them through many levels of abstraction in the `hibernate` library. The middle portion of the graph corresponds to the `Session.find` injection vector in `hibernate` discussed in Section 6.3.1. Another important observation about the connectivity graph in Figure 6.13 is that it is not uncommon to have *multiple paths* connecting the same source and sink pair. □

A connectivity picture similar to the case of `personalblog` is observed for `roller`, the only application in SecuriBench to suffer from false positives. However, in the case of `roller`, our allocation site inlining approach described in Section 3.4.4 is unable to sufficiently disambiguate one of tainted allocation site within `hibernate`. This illustrates how a single tainted object may lead to *multiple pairs* of connected sources and sinks, thereby increasing the false positive count.

When the vulnerability traces are reported to the user, it is generally useful to group traces by the source they start at as well as the sink they reach. This is because it is common for a single call to a sanitizer added to the code to get rid of a whole group of vulnerability traces that contain a particular source or a sink.

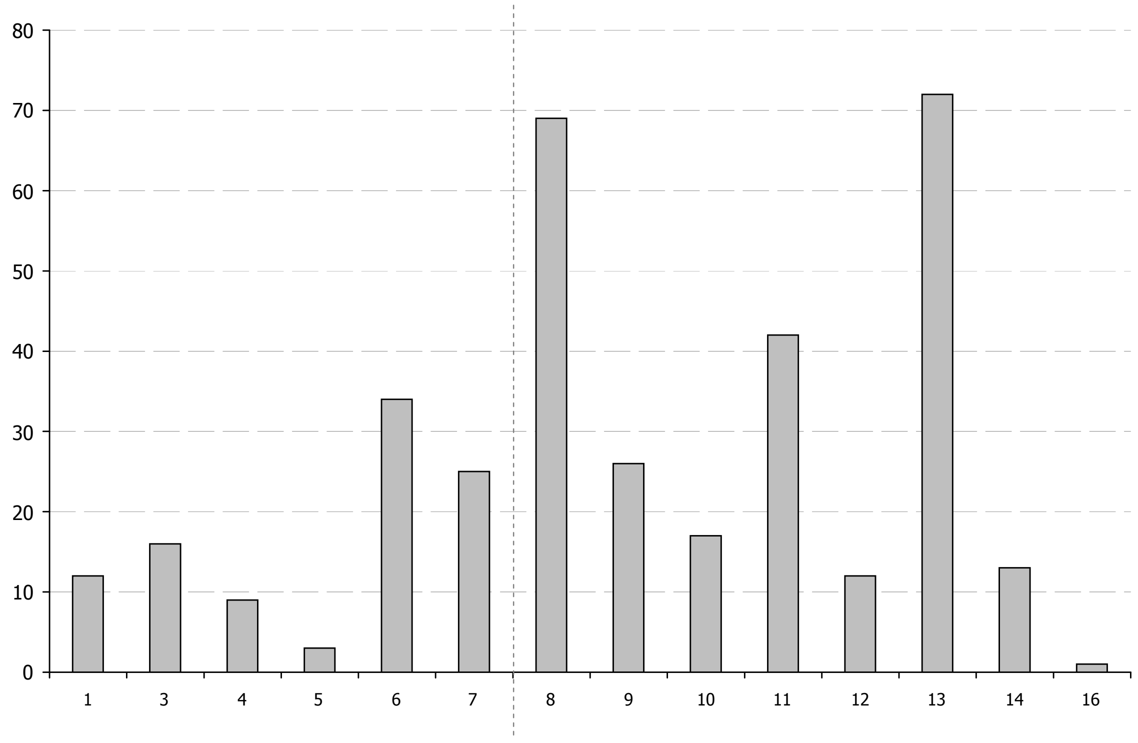


Figure 6.14: Vulnerability trace length histogram. The length is displayed on the x axis and the number of traces of this particular length is shown on the y axis. The dotted vertical line separates vulnerabilities that exist in application code only from those that span library code.

6.3.5 Shallow vs. Long Vulnerability Traces

The amount of work requited to find a vulnerability is roughly proportional to how long a particular vulnerability trace is. There is some controversy in the program analysis community as to how complex some of the bugs in real-life code are, which affects the sophistication of the analyses built to detect them. Some believe that relatively simple static analysis suffices for bug detection. In this section we report vulnerability trace statistics that shed additional light on this subject.

As described earlier, each vulnerability trace is a sequence of static object approximations $h_1, ..., h_n$. A single source-sink pair may correspond to several traces. Figure 6.14 shows a distribution of trace lengths for all the traces reported for 11 SecuriBench benchmarks. For the results presented in the histogram, we collected statistics for *all* such traces.

The first observation is that there are some really long traces our analysis manages to find, with 16 objects being the longest. Such a trace is truly difficult to find with either manual code inspection or a shallow analysis tool. Long traces usually span dozens of procedures as well as multiple files.

Another important observation is that most *long* traces are due to injection vectors located in library code, such as the SQL injection in `hibernate` described in Section 6.3.1. In fact, when we focus on non-library code only, the longest vulnerability trace located within application code is “only” 7 objects long. It would still be quite a challenge to find such a vulnerability trace by hand.

6.4 Runtime Analysis Results

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

The dynamic checker for the SQL injection query will match whenever a user controlled string flows in some way to a suspected sink, regardless of whether a user input is harmful in a particular execution. It will then react to replace the potentially dangerous string with a safe one. The PQL query is implemented as five separate state machines, one for each query. The effect of the instrumentation is to track all `String`s that either are directly user-controlled or that are derived from it, and to report a match if such a user-controlled string falls unsafely into Java’s SQL interface.

Note that even if a *given user input* is harmless in a particular execution, the data will still flow the same way, and thus will still be matched. The query does no direct checking of the value that has been provided by the user, so if harmless data is passed along a feasible injection vector, it will still trigger a match to the query. As a result of this, drastic responses such as aborting the application may not be suitable outside of a debugging context. Implementing a second level of checking that actually considers the values or just logging potentially malicious input as well as the injection

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

Figure 6.15: Summary of the number of instrumentation points, running times, dynamic overhead, both with and without optimizations. All times are given in seconds.

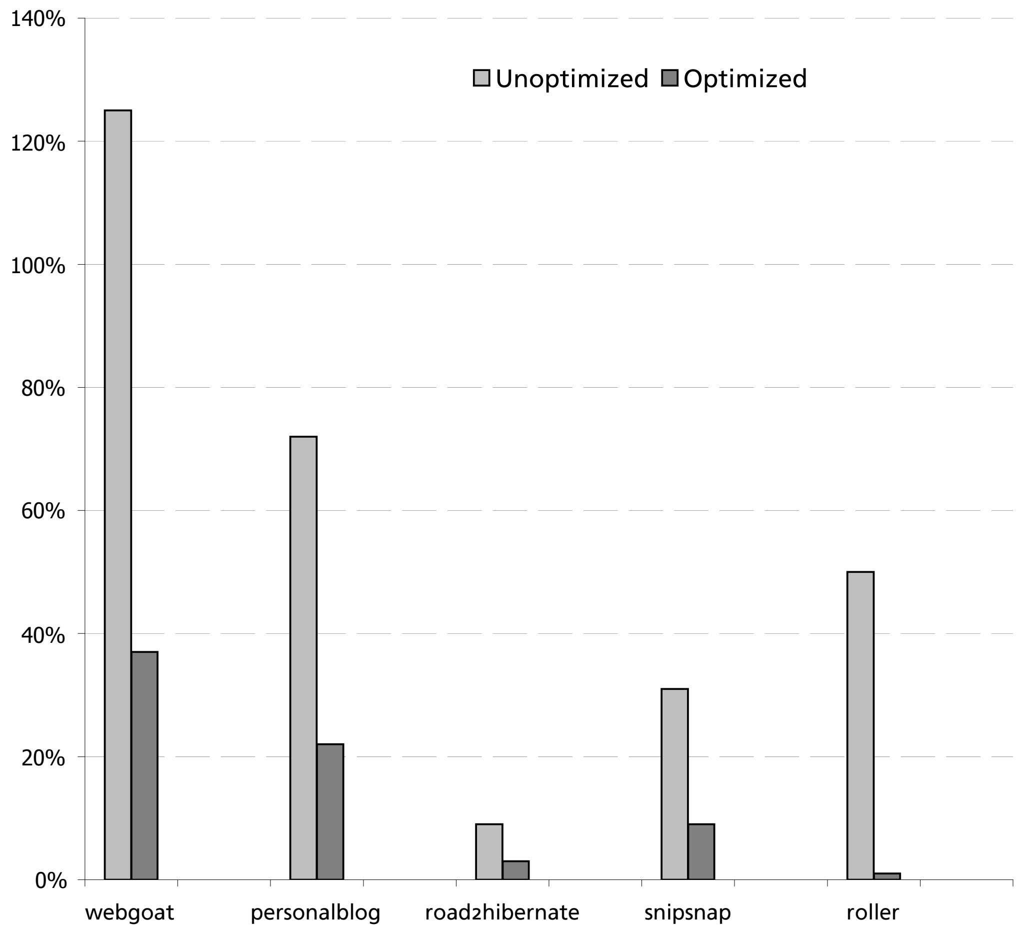


Figure 6.16: Runtime analysis overhead comparison.

paths may be appropriate. The rest of this section focuses on performance overhead incurred with different versions of our runtime instrumentation.

6.4.1 Performance Summary

Figure 6.15 summarized the runtime analysis overhead. Results are presented for both the unoptimized and the optimized runtime analysis versions. Several SecuriBench applications are missing from the table, as we were unable to install them for runtime analysis due to complex configuration and database dependency issues. Columns 2 and 3 show the number of instrumentation points that were inserted by the runtime instrumentation described in Chapter 5.

Columns 4 — 6 summarize the running times measured in seconds. Measuring Web application running times presents a number of unique challenges not present in command-line applications. The times we report for the Web applications reflect the average amount of time required to serve a *single* page in response to a single HTTP request, as measured by the standard profiling tool JMeter [56]. The only exception is `road2hibernate`, which is a command-line program and its time is a simple start-to-finish timing. Finally, columns 7 and 8 summarize the overhead with the unoptimized and optimized versions of the analysis.

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

6.4.2 Importance of Static Optimization

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

Fortunately, the static optimizer effectively removes instrumentation on calls to string processing routines that are provably not present on any path from user input to database access. Exploiting static information dramatically reduces both the number of instrumentation points and the overhead of the system, as shown in Figure 6.15. Figure 6.16 presents a graphical summary of runtime overhead results.

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

6.5 Chapter Summary

This chapter presents the experimental results of the Griffin project. We have applied our static and runtime techniques to two sets of benchmarks: large, open-source Java applications in common use as well as a set of small synthetic micro benchmark cases that exercise different analysis features. Our static technique finds almost one hundred vulnerabilities. The static analysis reported false positives for only one of 11 applications we have analyzed.

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.

While our runtime overhead can be quite high, information we compute statically allows us to reduce the number of necessary instrumentation points dramatically, reducing the dynamic overhead to below 10% in the majority of cases.

Chapter 7

Related Work

There has been an explosion of interest in practical analysis techniques for improving software reliability and security in recent years. Some of the important research highlights include the Instrinsa Prefix tool [28] that was later acquired by Microsoft and is now routinely used to find bugs in large code bases [118]. Other success stories include applying static analysis techniques to find bugs in operating systems code [16, 80, 93], Clouseau system for finding memory leaks [82], tools for finding buffer overruns [128, 208], FindBugs [86] for finding a variety of Java errors, etc. Runtime techniques have been applied to problems of debugging and reliability [81, 210] as well as software security, with StackGuard and Valgrind exemplifying some of the successes [19, 41, 43, 94, 153, 209].

The rest of this chapter is organized as follows. Section 7.1 summarizes techniques commonly used in the industry for improving Web application security posture. Section 7.2 outlines some relevant theoretical security frameworks. Sections 7.3 and 7.4 focus on static and runtime research for improving security, focusing both on Web application security issues as well as other types of attacks, such as buffer overruns. Section 7.5 focuses on some recent alternative approaches to Web application security that fall outside the domains of static and runtime analysis. Section 7.6 summarizes work related to reflection and call graph construction. Finally, Section 7.7 describes research related to PQL.

7.1 Web Application Security Techniques

There has been a great deal of interest in Web application security in recent years. This lead to the development of a range of technologies to address Web application vulnerabilities. This section introduces some of the more commonly used solutions. In addition to manual code reviews and client- or browser-side data validation [160], which are commonly employed for finding vulnerabilities, the two most commonly used approaches are *penetration testing* and *application firewalls*. These techniques have already been described extensively in Sections 1.4.2 and 1.4.3.

7.2 Security Frameworks

A considerable amount of theoretical research in language security exists, some of which is briefly summarized below.

7.2.1 Secure Information Flow

Much of the work in information-flow analysis uses a type-checking approach, as exemplified by JFlow [150]. 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. Security label polymorphism allows for code that is generic with respect to the security class of the data it manipulates.

The security type system in such a language enforces information-flow policies. Most information flow systems do not focus on the problem of type inference and focus on the enforcement problem instead. The annotation effort, however, may be prohibitively expensive in practice. In contrast, the focus of our static analysis is on finding violations of information flow properties without requiring annotations.

In addition to explicit information flows our approach addresses, JFlow also deals with implicit information flows through reverse channels [172].

7.2.2 Security Automata

Security automata have been widely investigated as a means of implementing security policies [174]. A security automaton enforces a security policy by monitoring the execution of a target system, and intercepting instructions which would otherwise violate the specified policy. For example, a user may specify that after a call to operation `FileRead`, the program may not call `Send`. The corresponding security automaton would monitor the target system, watching for calls to `FileRead`. If one was seen, the automaton would then monitor the system for an attempted call to `Send`. If such an attempt were made, it would intercept the call and execute error handling code instead. Conceptually, this is very similar to runtime error checking provided by PQL, even though the subquery mechanism of PQL makes it possible to capture context-free languages.

Walker uses security automata to encode security policies to be enforced in automatically generated code [196]. Erlingsson and Schneider use security automata to implement software fault isolation security policies, which prevent memory accesses outside of the allowable address space [49]. In that work, they discuss techniques used to merge security automata directly into binary code at the x86 assembler and Java Virtual Machine Language (JVML). Barker and Stuckey investigate role-based and temporal role-based access control policies, implemented using constraint logic specifications [17]. To the best of our knowledge, the notion of recovery achieved with our runtime analysis has not been previously explored in security automata literature.

7.3 Static Techniques for Security

A good overview of static analysis approaches applied to security problems is provided in Chess et al. [34]. 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 [206]. While a significant improvement on Unix `grep`, these tools, however, have no knowledge of how data propagates throughout the program and therefore cannot be used to automatically and fully solve taint-style problems.

7.3.1 Finding Vulnerabilities in Type-Unsafe Languages

Buffer overruns have long been the of programming in type-unsafe languages such as C and C++ for years. While they sometimes allow the hacker to completely take over the vulnerable system, they are often the enabling mechanism for other types of exploits, such as worms [149]. Some of the early static analysis approaches for finding buffer overruns include LCLint [51] and Splint [117], both of which incorporate type-based analysis techniques.

CSSV is another annotation-driven tool for buffer overrun detection [47]. Pre- and post-conditions of a function, are used to aid static analysis. CSSV converts the original C program into an integer program, where variables assume integer values and linear inequalities are generated from standard string manipulation routines. Correctness assertions that encode the validity of array accesses are explicitly included. Next, a conservative static analysis technique is used to detect faulty integer manipulations, which directly translate into bugs in the original code. The analysis is performed on a per-procedure basis, and annotations (called contracts) are used to make the analysis inter-procedural. The number of false alarms generated by the tool depends on the accuracy of the contracts. The analysis used by CSSV to check the correctness of integer manipulations was heavyweight and scaling it is a challenge.

An analysis approach that uses type qualifiers has been proven useful in finding security errors in C for the problems of detecting format string violations and user/kernel bugs in Linux device drivers [93, 177, 194]. A variety of C features contribute to a high false positive rate of this approach. Context sensitivity significantly reduces the rate of false positives encountered with this technique; however, it is unclear how scalable the context-sensitive approach is. However, this approach handles certain features of C, such as `unions`, and type casts unsoundly and also assumes memory safety, i.e. no buffer overruns.

In contrast to our technique, type qualifiers are *part of the source code*, whereas tainted object problem specifications in our framework are located on the side. Recently, there has been interest in extending type qualifier techniques to Java [68, 162]. Doing so would achieve the dual goal of having a more modular analysis and also having self-documenting code. The latter factor is especially important when it comes to newly developed code.

Ashcraft et al. use system-specific compiler extensions within the metacompilation framework to find a integer over- and underflows and user pointer dereferences [9]. Their paper uses a terminology of sources, sinks ,and sanitizers similar to that used in our framework described in Chapter 2. However, there is no explicit notion of derivation routines, as their focus is on programs written in C and manipulating characters explicitly. However, the semantics of `memcpy` and `bcopy` copying functions is taken into account. Their work also focuses on leveraging programming beliefs to find missing sources and sinks. We plan to use a similar approach, only at runtime to obtain more complete vulnerability specifications, as described in Section 8.2.2.

ARCHER targets unsafe memory and array accesses [208]. It works by symbolically simulating code execution, while maintaining information about variables in a database as the execution proceeds. At every array access, index values involved in the access are considered by a constraint solver in an effort to find potential violations. ARCHER also uses statistical code analysis to automatically infer the set of functions that it should track; this inference serves as a robust guard against omissions, especially in large systems which can have hundreds of such functions.

Livshits et al. present a program representation designed for bug detection called IPSSA [128]. A hybrid pointer analysis that tracks actively manipulated pointers held in local variables and parameters accurately with path and context sensitivity and handles pointers stored in recursive data structures less precisely but efficiently is proposed. IPSSA unsoundly assumes no aliasing between pointers reached from function parameters. It has been successfully applied to find a number of buffer overruns with very few false positives, however, scaling the analysis often presents a challenge for larger benchmarks.

A pointer analysis of Avots et al. is used to find a number of buffer overruns and format string violations statically [10]. The pointer analysis described there uses the same Datalog- and `bddbddb`-based approach described in Chapter 3. A key contribution of this work is that, in addition to a conservative analysis, it proposes an optimistic analysis that assumes a more restricted C semantics that reflects common C usage to increase the precision of the analysis. In a manner similar to our static/runtime interplay, the analysis described in Avots et al. is also effective at reducing the overhead of the CRED compiler, a previously proposed runtime prevention technique for memory errors [94].

7.3.2 Static Analysis for Web Application Security

Languages such as C and C++ are not common for Web application development. As a result, static approaches for improving Web application security have focused on type-safe languages such as Java and also scripting languages such as PHP.

Analyzing Type-Safe Languages

One particular problem in the Web application security space that has attracted much attention is that of SQL injections. Static analysis has been applied to analyzing SQL statements constructed in Java programs that may lead to SQL injection vulnerabilities [36, 66, 67, 198]. The underlying technique is to analyze 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. Since accurate pointer information is necessary to construct an accurate flow graph, at this point it is unclear whether this technique can achieve the scalability and precision needed to detect errors in large systems.

Our static analysis approach represents the first comprehensive technique to address the issue of flow graph construction. We believe that more precise string analysis can be added on top of the string provenance results obtained with our approach to obtain better string approximations for a range of applications.

Analyzing Scripting Languages

Recently, there has been an increased interest in static analysis of scripting languages such as PHP. The WebSSARI project pioneered this line of research. WebSSARI uses combined unsound static and dynamic analysis in the context of analyzing PHP programs [88]. WebSSARI has successfully been applied to find many SQL injection and cross-site scripting vulnerabilities in PHP code. However, while practical and often quite effective, WebSSARI is also unsound. Creating a precise *sound* analysis for scripting languages presents a significant challenge in the presence of the `eval` construct that allows arbitrary code to be executed.

A limitation of WebSSARI is its analysis power: (1) the analysis is intraprocedural and does not infer function pre- and post-conditions, thus requiring extensive annotations to use; (2) it does not model predicates and conditional branches, which is a key mechanism for testing and sanitizing input variables in PHP; and (3) it uses a generic type based algorithm which does not model dynamic features in scripting languages like PHP. For example, dynamic typing may introduce subtle errors that WebSSARI misses. The include statement dynamically inserts code to the program which may contain, induce, or prevent errors.

Several projects that came after WebSSARI improve on the quality of static analysis used for analyzing PHP code. Xie et al. use a architecture enables us to handle dynamic features unique to scripting languages such as dynamic typing and code inclusion, which have not been adequately addressed before [207]. However, recursive function calls are simply ignored instead of being handled correctly. Moreover, only local alias analysis is performed, which likely contributes to the number of false negatives incurred with this approach. Multi-dimensional arrays also appear to be unsupported. Xie et al. apply a heuristic for resolving simple cases of include statements that seems to yield good results in practice.

The Pixy project [95, 97] implements a more ambitious flow-sensitive, interprocedural, and context-sensitive data flow analysis technique for PHP. As mentioned before, unlike Java, PHP generally requires a flow-sensitive analysis approach. Additional literal analysis (a form of copy and constant propagation) and alias analysis [96] steps performed by Pixy lead to more comprehensive and precise results than those provided by previous approaches.

7.4 Runtime Analysis for Security

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

7.4.1 Finding Vulnerabilities in Type-Unsafe Languages

A range of compiler extensions discussed below has been used to protect against memory-based attacks prevalent in C programs such as format string violations and buffer overruns. A good overview of these techniques is given in Kc et al. [98].

FormatGuard, a compiler modification, injects code to dynamically check and reject all `printf`-like function calls where the number of arguments does not match the number of "%" specifiers in the format string [42]. Of course, only applications that are re-compiled using FormatGuard will benefit from its protection. Also, one technical shortcoming of FormatGuard is that it does not protect user-defined wrappers for the `printf` family of routines. An unfortunate consequence of the design choices of FormatGuard is that programs with format string vulnerabilities remain vulnerable to denial of service attacks.

A wide range of approaches focuses on runtime buffer overrun protection. Products such as StackGuard [41], StackShielf [8] and the `/GS` switch implemented in the later version of the Microsoft Visual Studio compilers [39] all use similar techniques to provide protection against stack smashing exploits. StackGuard works by placing a "canary" word next to the return address on the stack. If the canary word has been altered when the function returns, then a stack smashing attack has been attempted while within the function. The StackGuard-protection program responds by emitting an intruder alert and then halting the program. Unfortunately, while generally effective, this sort of stack protection can still be circumvented with more sophisticated attack techniques such as spoofing the canary, etc. [169, 27].

PointGuard focuses on heap-based buffer overrun exploits [43]. PointGuard-protected programs encrypts all pointers while they reside in memory and decrypts them only before they are loaded to a CPU register. Similarly to FormatGuard and

StackGuard, PointGuard is implemented as an extension to the GCC compiler, which injects the necessary instructions at compilation time, allowing a pure-software implementation of the scheme. The overhead incurred with PointGuard may, however, be prohibitively expensive [191].

Kiriansky et al. propose program shepherding, a policy-driven mechanism for closely monitoring and dynamically controlling the flow of program execution [101]. The advantage of program shepherding is that the original program does not need to be recompiled. They define different default and customizable security policies for code based on the nature of its origin, whether it was loaded from the local file system, generated by the running program itself, or if it self-mutated. Their system is integrated into an interpreter, which enables the sandboxed checking of running applications and monitoring of their control-flow. While the functionality of this approach is attractive, the fact that it is interpreted makes for significant overhead.

7.4.2 Runtime Analysis for Web Application Security

Scott et al. present a structuring technique which helps designers abstract security policies from large Web applications [176]. Their system consists of a specialized Security Policy Description Language which is used to program an application-level firewall. Security policies are written and compiled for execution on the security gateway. The security gateway dynamically analyses and transforms HTTP requests and responses to enforce the specialized policy. To the best of our knowledge, this system has not been applied to large Web applications.

Protection from SQL Injections

Several techniques focus on SQL injections exclusively. Buehrer et al. propose a technique that is based on comparing, at execution time, the parse tree of the SQL statement *before* inclusion of user input with that resulting *after* the inclusion of user-provided input [26]. SQLRand used SQL keyword randomization in order to create SQL language keywords that are not easily guessable by the attacker, thus foiling most SQL injection techniques that involve adding extra SQL commands [22].

AMNESIA is a model-based approach that detects illegal queries before they are executed on the database [76, 77, 78, 79]. In its static part, the technique uses program analysis to automatically build a model of the legitimate queries that could be generated by the application. In its dynamic part, this technique uses runtime monitoring to inspect the dynamically-generated queries and check them against the statically-built model. Depending on the quality of the statically-derived model, their technique may suffer from both false positives and false negatives. Moreover, it is unclear how their static analysis would scale to large programs, as it has only been evaluated with relatively small benchmarks.

Dynamic Taint Propagation

Dynamic taint propagation described in Haldar et al. borrows much from our runtime technique [75]. In contrast to our technique, they use heuristics similar to those use in the Perl taint mode [197] to determine which `String`s need to be untainted at runtime; i.e. matching against regular expressions is assumed to be an untainting operation. However, unlike the runtime techniques of the Griffin project described in Chapter 5, their approach is unable to provide recovery from vulnerabilities.

Pietraszek et al. propose CSSE, a system that modifies the PHP interpreter to tag strings to distinguish those that are developer-supplied from those that are provided as input. Since CSSE tracks where the different segments of a string originate, it is able to provide user string escaping or recovery in a manner similar to that of our runtime technique. Su et al. describe SqlCheck, a similar system for SQL injection detection that works on both Java and PHP code [183]. SqlCheck has been shown effective at preventing SQL injections in a range of medium-sized Web applications.

PHPrevent is a project that focuses on securing PHP applications [154]. While similar in spirit to our runtime protection described in Chapter 5, PHPrevent uses a modified PHP interpreter to precisely track taint at runtime. Unlike our approach, however, the granularity of taint tracking is greater: tainting is recorded and propagated at the level of individual characters. Their approach to untainting is to escape parts of the input contained in the output. However, their notion of white-listing the allowed input is somewhat arbitrary and will not necessarily work for applications such bulletin boards that require some of the HTML tags to pass through. This is not unlike our notion of built-in sanitizers discussed in Sections 5.5.1 and 5.5.2.

7.5 Other Web Application Security Approaches

Several techniques that fall outside the realm of static and runtime language-based vulnerability detection are described below.

7.5.1 Intrusion Detection

Kruegel et al. describe several intrusion detection systems that use a variety of different anomaly detection techniques to detect attacks against Web servers and Web-based applications [114, 115, 193]. These systems analyze client queries that reference server-side programs and creates models for a wide-range of different features of these queries. Examples of such features are access patterns of server-side programs or values of individual parameters in their invocation. As with other types of intrusion detection techniques, proper patterns can be learned from prior "training" traffic.

In particular, the use of application-specific characterization of the invocation parameters allows the system to perform focused analysis and produce a reduced number of false positives. The system automatically derives parameter profiles associated with Web applications (e.g., length and structure of parameters) and relationships between queries (e.g., access times and sequences) from the analyzed data. Therefore, it can be deployed in very different application environments without having to perform time-consuming tuning and configuration. However, unlike the techniques in Section 7.4 intrusion detection-based schemes cannot provide strong guarantees on which vulnerabilities are detected and which are missed.

7.5.2 Client-side Protection

RequestRodeo partly disables the inclusion of authentication information into requests passed to the server [92]. A proxy that resides between the browser and the application server identifies HTTP requests which qualify as potential Cross Site Request Forgery attacks and strips them from all possible authentication credentials. RequestRodeo is implemented in the form of a proxy instead of integrating it directly into a Web browser to provide protection for a variety of Web browsers.

Noxes, another browser-based technology, is designed to protect against information leakage from the user's environment while requiring minimal user interaction and customization effort [100]. For instance, the act of sending the cookie information to an unknown URL will be detected and the user will be prompted whether this action should continue. Information leakage is a frequent side-effect of cross-site scripting attacks.

7.5.3 System Design for Better Application Security

Tahoma is a virtual machine-based execution framework for Web browsers and applications [44]. It provides a level of isolation between Web applications and the underlying operating systems and allows limiting the capabilities of individual applications. For instance, the set of URLs available to a particular application may be restricted in Tahoma by the application publisher.

Some of the same ideas have been explored in Terra, a flexible architecture for trusted computing, called Terra, that allows applications with a wide range of security requirements to run simultaneously on commodity hardware [62]. Applications on Terra enjoy the semantics of running on a separate, dedicated, tamper-resistant hardware platform, while retaining the ability to run side-by-side with normal applications on a general purpose computing platform. Terra achieves this synthesis by use of a trusted virtual machine monitor (TVMM) that partitions a tamper-resistant hardware platform into multiple, isolated virtual machines (VM), providing the appearance of multiple boxes on a single, general-purpose platform.

7.6 Reflection and Call Graph Construction

General treatments of reflection in Java are given in Forman and Forman [54] and Guéhéneuc et al. [74]. The rest of the related work falls into the following broad categories: projects that explicitly deal with reflection in Java and other languages; approaches to call graph construction in Java; and finally, static and dynamic analysis algorithms that address the issue of dynamic class loading.

7.6.1 Reflection and Metadata Research

The metadata and reflection community has a long line of research originating in languages such as Scheme [186]. We only mention a few highly relevant projects here. The closest static analysis project to ours we are aware of is the work by Braux and Noyé on applying partial evaluation to reflection resolution for the purpose of optimization [24]. Their paper describes extensions to a standard partial evaluator to offer reflection support. The idea is to "compile away" reflective calls in Java programs, turning them into regular operations on objects and methods, given constraints on the concrete types of the object involved. The type constraints for performing specialization are provided by hand.

Our static analysis can be thought of as a tool for inferring such constraints, however, as our experimental results show, in many cases targets of reflective calls cannot be uniquely determined and so the benefits of specialization to optimize program execution may be limited. Braux and Noyé present a description of how their specialization approach may work on examples extracted from the JDK, but lacks a comprehensive experimental evaluation. In related work for languages other than Java, Ruf explores the use of partial evaluation as an optimization technique in the context of CLOS [171]. Masuhara et al. explore the use of partial evaluation as applied to an abstract object-oriented language [143].

The issue of *specifying* reflective targets is explicitly addressed in Jax [188]. Similarly, the Spark pointer analysis implemented within the Soot compiler uses specifications of many reflective targets in the JDK during call graph construction [121].

Just like our technique, Spark also used on-the-fly call graph construction. Potential reflective call targets are automatically added to the set of root methods in the beginning of the analysis. Unlike Spark, which comes with models of many `native` methods, our approach is oblivious to `native` routines. While this is generally unlikely, not handling such methods can render our points-to results not fully sound. Handling of `native` methods in Java is addressed in Zhang et al. [212].

Jax is concerned with reducing the size of Java applications in order to reduce download time; it reads in the class files that constitute a Java application, and performs a whole-program analysis to determine the components of the application that must be retained in order to preserve program behavior. Clearly, information about the true call graph is necessary to ensure that no relevant parts of the application are pruned away. Jax's approach to reflection is to employ user-provided specifications of reflective calls. While our framework also supports user-provided annotations, as illustrated in Section 4.6.4, determining targets of reflective calls can often be error-prone, if delegated to the user. To assist the user with writing complete specification files, Jax relies on dynamic instrumentation to discover the missing targets of reflective calls. Our analysis based on points-to information can be thought of as a tool for determining where to insert reflection specifications.

A precise analysis of strings by Christensen et al. mentions reflections as one of the potential uses of their approach [36]. They treat `Class.forName` calls as "hotspots" for their analysis, then trying to determine what the exact values passed as parameters may be. Their approach, however, relies on an external pointer analysis to determine the propagation of strings throughout the program. The paper applies their approach to programs that are all under 4,000 lines long and lacks a detailed experimental evaluation of the precision of their approach. Their technique, however, can potentially address reflective calls that have much more complex string expressions passed as reflective arguments. For example, knowing that the argument of `Class.forName` must end in string `"Configuration"` will allow the analysis to substantially limit the number of possibly instantiated classes.

7.6.2 Call Graph Construction

Much effort has been spent of analyzing function pointers in C [48, 144, 147] as well as virtual method calls in C++ [3, 12, 30, 161] and Java [71, 72, 166, 184, 189]. They are described in more detail below.

Function Pointers in C

Emami et al. describe how a context-sensitive pointer analysis for C integrated with call graph construction in the presence of function pointers [48]. Their approach introduces the notion of call graph discovery when the call graph is unavailable in advance.

Milanova et al. evaluate the precision of call graph construction in the presence of function pointers using an inexpensive pointer analysis approach [211] and conclude that it is sufficient for most cases [144, 147].

Virtual Calls in C++

Bacon et al. compare the “unique name”, RTA, and CHA virtual call resolution approaches [12, 11]. They conclude that RTA is both fast and effective and able to resolve 71% of virtual calls on average.

Aigner and Hölzle investigate the effect virtual call elimination using CHA has on the runtime of large C++ programs and report a median 18% performance improvement over the original programs [3]. The number of virtual function calls is reduced by a median factor of five.

Virtual Calls in Java

Grove et al. present a parameterized algorithmic framework for call graph construction [71, 72]. They empirically assess a multitude of call graph construction algorithms by applying them to a suite of medium-sized programs written in Cecil and Java. Their experience with Java programs suggests that the effect of using context sensitivity for the task of call graph construction in Java yields only moderate improvements.

Tip and Palsberg propose a propagation-based algorithm for call graph construction and investigate the design space between existing algorithms for call graph construction such as 0-CFA and RTA, including RA, CHA, and four new ones [189]. Sundaresan et al. go beyond the tranditional RTA and CHA approaches in Java and and use type propagation for the purpose of obtaining a more precise call graph [184]. Their approach of using variable type analysis (VTA) is able to uniquely determine the targets of potentially polymorphic call sites in 32% to 94% of the cases.

Agrawal et al. propose a demand-driven algorithm for call graph construction [1]. Their work is motivated by the need for just-in-time or dynamic compilation as well as program analysis used as part of software development environments. They demonstrate that their demand-driven technique has the same accuracy as the corresponding exhaustive technique. The reduction in the graph construction time depends upon the ratio of the cardinality of the set of influencing nodes to the set of all nodes.

Rayside et al. explore the effect various call graph construction techniques have on automatic clustering approaches used to extract the high level structure of the program under study [166]. They also used a slightly different notion of the call graph that supports weighted edges.

7.6.3 Dynamic Analysis Approaches

Our work is motivated to a large extent by the need of error detection tools to have a static approximation of the true conservative call graph of the application. This largely precludes dynamic analysis that benefits optimizations such as method inlining and connectivity-based garbage collection.

A recent paper by Hirzel, Diwan, and Hind addresses the issues of dynamic class loading, native methods, and reflection in order to deal with the full complexity of Java in the implementation of a common pointer analysis [85]. Their approach involves converting the pointer analysis [5] into an online algorithm: they add constraints between analysis nodes as they are discovered at runtime. Newly generated constraints cause re-computation and the results are propagated to analysis clients such as a method inliner and a garbage collector at runtime. Their approach leverages the class hierarchy analysis (CHA) to update the call graph. Our technique uses a more precise pointer analysis-based approach to call graph construction.

Their paper also contains a comprehensive overview of analysis approaches that address dynamic class loading. Here we briefly mention some of the highlights. However, none of the projects mentioned below fully address the issue of reflection.

The ARE tool presented in Gscwind et al. allows tracing of method parameter and return values at runtime for program comprehension [73]. They point out that ignoring reflection leads to program traces that are incomplete. ARE instruments the program and collects data that allows it to provide targets of reflective method calls. These reflective targets are subsequently displayed by way of sequence diagrams.

Pechtchanski and Sarkar [163] present a framework for interprocedural whole-program analysis. They discuss how the analysis is triggered (when newly loaded methods are compiled), and how to keep track of what to de-optimize (when optimistic assumptions are invalidated). Qian and Hendren [165] adapt Tip and Palsbergs XTA [189] to deal with dynamic class loading. The main contribution of their work is a low-overhead call edge profiler, which yields a precise call graph upon which XTA is based.

7.7 PQL and Runtime Matching Formalisms

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

7.7.1 Aspect-Oriented Formalisms

PQL attaches user-specified actions to subquery matches; this capability puts PQL in the class of *aspect-oriented* programming languages [99, 159]. Maya [14] and AspectJ [99] attach actions based on syntactic properties of individual statements in the source code. The DJ system defines aspects as traversals over a graph representing the program structure [159].

PQL system may be considered as an aspect-oriented system that defines its aspects with respect to the dynamic history of sets of objects. An extension of AspectJ to include “dataflow pointcuts” has been proposed to represent a statement that receives a value from a specific source. PQL can represent these with a two-statement query, and permits much more complex concepts of data flow [142]. Walker and Veggers introduce the concept of *declarative event patterns*, in which regular expressions of traditional pointcuts are used to specify when advice should run [196]. Allan et al. extend this further by permitting PQL-like free variables in the patterns [4]. PQL differs from these systems in that its matching machinery can recognize non-regular languages, and in exploiting advanced pointer analysis to prove points irrelevant to eventual matches.

7.7.2 Other Program Query Languages

Systems like ASTLOG [45] and JQuery [91] permit patterns to be matched against source code; Liu et al. [124] extend this concept to include parametric pattern matching [13]. These systems, however, generally check only for source-level patterns and cannot match against widely-spaced events. A key contribution of PQL is a pattern matcher that combines object-based parametric matching across widely-spaced events. Lencevicius et al. developed an interactive debugger based on queries over the heap structure [119]. This analysis approach is orthogonal both to the previous systems named in this section as well as to PQL; however, like PQL, its query language is explicitly designed to resemble code in the language being debugged.

The Partiqle system [65] uses a SQL-like syntax to extract individual elements of an execution stream. It does not directly combine complex events out of smaller ones, instead placing boolean constraints between primitive events to select them as sets directly. Variables of primitive types are handled easily by this paradigm, and nearly arbitrary constraints can be placed on them easily, but strict ordering constraints require many clauses to express.

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

7.7.3 Analysis Generators

PQL follows in a tradition of powerful tools that take small specifications and use them to automatically generate analyses. Metal [80] and SLIC [15] both define state machines with respect to variables. These machines are used to configure a static analysis that searches the program for situations where error transitions can occur. Metal restricts itself to finite state machines, but has more flexible event definitions and can handle pointers (albeit in an unsound manner).

The Rhodium language [120] uses definitions of dataflow facts combined with temporal logic operators to permit the definition of analyses whose correctness may be readily automatically verified. As such, its focus is significantly different from the other systems, as its intent is to make it easier to directly implement correct compiler passes than to determine properties of or find bugs in existing applications. Likewise, though it is primarily intended as a vehicle for predefined analyses, Valgrind [153] also presents a general technique for dynamic analyses on binaries.

Chapter 8

Conclusions and Future Work

This chapter is organized as follows. Section 8.1 below summarizes the contributions made by this thesis. Section 8.2 outlines some of the future research directions.

8.1 Thesis Contributions

This thesis describes a solution to the practical problem of Web application security. Web application security vulnerabilities such as SQL injection and cross-site scripting attacks now account for the majority of all application security issues. Commonly used solutions such as client-side sanitization, penetration testing, and application firewalls do not provide an adequate solution to these problems.

The Griffin project described in this thesis provides a hybrid static and runtime analysis solution that addresses a wide range of Web application vulnerabilities. The analysis framework the Griffin project allows the user to specify which vulnerabilities they are looking for. The vulnerability specification are expressed in a general language called PQL, which, while a generic language for describing events on objects, is well-suited to the task of specifying vulnerabilities.

Our static contribution is in enhancing the precision of points-to information. While the scalable BDD-based context-sensitive analysis our technique is based on is quite precise, it does not handle many common patterns present in real programs such as containers and factory function. More precise allocation site handling used by our approach allows us to gain extra precision. In addition to this, precision is obtained by handling maps more precisely. The static analysis approach is sound modulo specification completeness and correctness.

Our static technique pushes the state of the art in pointer analysis precision. Moreover, our static analysis work constitutes one of the first uses of a context-sensitive pointer analysis on large programs. The soundness of our technique is another feature that sets it apart from much work in bug detection. Our static approach constitutes an important contribution to the area of Web application security. With the advent of Web application security vulnerabilities, software security now concerns more developers than ever before. Our contribution in this area raises the hope that one day Web application vulnerabilities described here will become a thing of the past.

In addition to the core static analysis technique, this thesis also proposes an analysis of reflection that allows us to obtain much large static call graphs. This is the first time the issue of reflection is explicitly addressed by a call graph construction algorithm. Our approach to reflection has since been adapted by others [53].

Our runtime contribution consists of designing a system for preventing vulnerabilities at runtime by maintaining precise and customizable dynamic taint data. What makes our runtime system unique is that it allows recovery from vulnerabilities by applying user-provided sanitizers on execution paths that do not have them already.

Our extensive experimental evaluation validates the effectiveness of Griffin static and runtime techniques. We formulated a variety of widespread vulnerabilities including SQL injections, cross-site scripting, HTTP splitting attacks, and other types of common Web application security issues and applying our static and runtime techniques to 11 large open-source J2EE applications.

Our experimental results show that our analysis is an effective practical tool for finding security vulnerabilities. We were able to find a total of 98 security errors, and all but one of our 11 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 static analysis reported false positives for only one of 11 applications we have analyzed. While our runtime overhead can be quite high, information we compute statically allows us to reduce the number of necessary instrumentation points dramatically, reducing the dynamic overhead to below 10% in the majority of cases.

8.2 Future Work

This section outlines several extensions for static and runtime analysis techniques described in earlier chapters. These extensions have the potential to become fruitful future research directions.

8.2.1 Better Static Analysis Scalability

As described in Section 3.4.3 the default scalable context-sensitive analysis algorithm is insufficient for our static analysis in practice. An object sensitive analysis as specified in Milanova et al. [145, 146] does not appear to scale to programs larger than 300 classes. While we have implemented a practical solution as described in Section 3.4.4 this solution has shortcomings and involves manual involvement to keep the false positives at bay.

A scalable object-sensitive pointer analysis would be a major boost for analysis precision in object-oriented languages. It is believed that object sensitivity is a much better match for languages such as Java, compared to the notion of context sensitivity, which came from languages such as C. Milanova et al. [145] have demonstrated that object sensitivity archives better precision compared to context sensitivity for a range of problems and our own experience has been similar.

A possible brute-force approach to better scalability of BDD-based analyses consists of parallelizing the problem. A major scalability bottleneck for Datalog style of program analysis is the need to come up with a *variable order* [203]. If the variable order is not suitable for the problem at hand, the solver usually runs our of memory quickly or iterates for a long time. A great deal of work goes into choosing a suitable variable order, however, a *global* variable order is in a sense a compromise.

If we are able to analyze the dependencies between the rules within the Datalog query and then break up the rules into groups so that the inter-group dependencies are minimized, we will be able to select much more suitable variable orders for individual rule groups. Each group of rules would be executed on a separate processors and will complete much faster because the variable order on each processor had been optimized to the rules being executed. A similar approach for hardware verification has been shown to work in Iyer et al. [90].

Another approach to achieving better scalability is using a more modular technique such as a type analysis. LCLint used type annotation for security purposes [51]. CQual has been used successfully to find taint-style vulnerabilities using a type qualifier-based approach [55]. Recently, there has been interest in incorporating a type qualifier-based approach to Java [68, 162].

8.2.2 Specification Discovery

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 finds *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.

A fruitful research direction is data mining techniques that observe the transformation of a single piece of data in order to infer the proper processing steps [148].

8.2.3 Model Checking

Model checking can be used in several ways as applied to vulnerabilities in Web-based applications. While using a pointer analysis allows our approach to tracking tainting values to scale well, it also makes the analysis results more difficult to explain. A vulnerability trace consists of a sequence of static heap object approximations. These are essentially allocation sites, and the original line number of information is given to the user, as long as it has been preserved in the original bytecode. However, when presented with two sequential allocation sites that may be located far apart, deciding how exactly they are connected may be a challenge, as that can involve multiple levels of parameter passing and method returns, all details that are abstracted away by the pointer analysis and thus do not make it into our analysis representation.

However, it is variables, not heap allocation sites that the developer typically reasons about. Even if we use the pointer analysis results to map from a heap location *back* to variables, there may be a multitude of variables corresponding to the same heap location; this might make result interpretation even more complex.

One approach to explaining the results is to use guided model checking. A static vulnerability trace discovered by static analysis may be used by the model checker to find the next "hop" to jump to. For example, if the statically determined vulnerability trace has heap object h_1 in method m_1 and h_2 in method m_2, then once method m_2 is accessed after m_1 during execution, alternative ways to get to m_2 will not be considered. This considerably reduces the scope of the search that needs to be done by the model checker. Model checking may be used to validate the feasibility of the statically obtained results. It is possible for our technique to generate false positives, when, for example, predicates matter. A model checker would be able to assertain the validity of a statically detected result. Model checking can be also used as a controlled execution environment for a developer to examine the located vulnerabilities. Since the values of all variables can be recorded into "snapshots", the developer should be able to go back in forth through a dangerous code path until he or she is certain of what is going on. This is considerably better than repeatedly rerunning an application only to hit the necessary breakpoint in the debugger.

Finally, a symbolic execution technique such as DART [64] or EXE [29] may be used to automatically derive exploits to statically discovered vulnerabilities. Static analysis tools often have to compete with customer-reported bugs that need to be urgently fixed. Having an exploit scenario would propel a statically discovered vulnerability from the realm of possibility to a real issue requiring urgent attention.

Bibliography

[1] Gagan Agrawal, Jinqian Li, and Qi Su. Evaluating a demand driven technique for call graph construction. In *Proceedings of the International Conference on Compiler Construction*, pages 29–45, April 2002.

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

[3] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 142–166, July 1996.

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

[5] Lars O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.

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

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

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

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

[10] Dzintars Avots, Michael Dalton, Benjamin Livshits, and Monica S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the International Conference on Software Engineering*, May 2005.

[11] David F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, University of California at Berkeley, May 1998.

[12] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–341, October 1996.

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

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

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

[16] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 1–3, January 2002.

[17] Steve Barker and Peter J. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Transactions of Information Systems Security*, 6(4):501–546, 2003.

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

[19] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanović, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the Conference on Computer and Communications Security*, October 2003.

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

[21] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 103–114, June 2003.

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

[23] John Boyland. Alias burying: unique variables without destructive reads. *Softw. Pract. Exper.*, 31(6):533–553, 2001.

[24] Mathias Braux and Jacques Noyé. Towards partially evaluating reflection in Java. In *Proceedings of the Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 2–11, January 1999.

[25] Jason Brittain and Ian F. Darwin. *Tomcat: the Definitive Guide*. O'Reilly and Associates, 2003.

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

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

[28] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software - Practice and Experience*, 30:775–802, 2000.

[29] Cristian Cadar and Dawson Engler. Execution generated test cases: how to make systems code crash itself. In *Proceedings of the International SPIN Workshop on Model Checking of Software*, August 2005.

[30] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 397–408, January 1994.

[31] Chuck Cavaness. *Programming Jakarta Struts*. O'Reilly Media, 2004.

[32] Cenzic, Inc. Hailstorm. http://www.cenzic.com.

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

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

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

[36] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the International Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 1–18, June 2003. Available from http://www.brics.dk/JSA/.

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

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

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

[40] Symantec Corporation. Internet security threat report. http://www.symantec.com/enterprise/threatreport/index.jsp, 2005.

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

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

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

[44] Richard S. Cox, Steven D. Gribble, Henry M. Levy, and Jacob Gorm Hansen. A safety-oriented platform for Web applications. In *Proceedings of the Symposium on Security and Privacy*, pages 350–364, May 2006.

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

[46] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952:77–101, 1995.

[47] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2003.

[48] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[49] Ulrih Erlingsson and Fred Schneider. SASI enforcement of security policies: a retrospective. In *Proceedings of the New Security Paradigms Workshop*, September 2000.

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

[51] David Evans. Static detection of dynamic memory errors. In *Proceedings of the Conference on Programming Language Design and Implementation*, May 1996.

[52] Fielding, et al. HTTP/1.1, Internet RFC 2616. `http://www.w3.org/Protocols/rfc2616/rfc2616.html`, 1999.

[53] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 133–144, July 2006.

[54] Ira R. Forman and Nate Forman. *Java Reflection in Action.* Manning Publications, 2004.

[55] Jeffrey Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceddings of the Conference on Programming Language Design and Implementation*, pages 1–12, June 2002.

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

[57] Apache Foundation. The byte code engineering library. http://jakarta.apache.org/bcel/, 2006.

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

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

[60] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 62–72, June 2005.

[61] Michael Furr and Jeffrey S. Foster. Polymorphic type inference for the JNI. Technical Report CS-TR-4759, University of Maryland, Computer Science Department, November 2005.

[62] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the Symposium on Operating Systems Principles*, October 2003.

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

[64] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 213–223, June 2005.

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

[66] Carl Gould, Zhendong Su, and Premkumar Devanbu. JDBC checker: a static analysis tool for SQL/JDBC applications. In *Proceedings of the International Conference on Software Engineering*, pages 697–698, May 2004.

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

[68] David Greenfieldboyce and Jeffrey S. Foster. Type qualifiers for Java. http://www.cs.umd.edu/Grad/scholarlypapers/papers/greenfiledboyce.pdf, August 2005.

[69] 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.

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

[71] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions of Programming Language Systems*, 23(6):685–746, 2001.

[72] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 108–124, October 1997.

[73] Thomas Gschwind, Johann Oberleitner, and Martin Pinzger. Using run-time data for program comprehension. In *Proceedings of the International Workshop on Program Comprehension*, page 245, May 2003.

[74] Yann-Gaël Guéhéneuc, Pierre Cointe, and Marc Ségura-Devillechaise. Java reflection exercises, correction, and FAQs. `http://www.yann-gael.gueheneuc.net/Work/Teaching/Documents/Practical-ReflectionCourse.doc.pdf`, 2002.

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

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

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

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

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

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

[81] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, pages 291–301, 2002 2002.

[82] David Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceddings of the Conference on Programming Language Design and Implementation*, pages 168–181, June 2003.

[83] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 254–263, June 2001.

[84] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering*, Snowbird, UT, June 2001.

[85] Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading. In *Proceedings of the European Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 96–122, June 2004.

[86] David Hovemeyer and William Pugh. Finding bugs is easy. In *Proceedings of the Onward! Track of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2004.

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

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

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

[90] Subramanian K. Iyer, Jawahar Jain, Debashis Sahoo, and Takeshi Shimizu. Verification of industrial designs using a computing grid with more than 100 nodes. In *Proceedings of the Asian Test Symposium*, December 2005.

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

[92] Martin Johns and Justus Winter. RequestRodeo: client side protection against session riding. In *Proceedings of the OWASP AppSec Europe Conference*, May 2006.

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

[94] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the International Workshop on Automatic Debugging*, pages 13–26, May 1997.

[95] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities (short paper). In *Proceddings of the Symposium on Security and Privacy*, May 2006.

[96] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of Web application vulnerabilities. In *Proceedings of the Workshop on Programming Languages and Analysis for Security*, June 2006.

[97] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for syntactic detection of Web application vulnerabilities. In *Proceedings of the Workshop on Programming Languages and Analysis for Security*, June 2006.

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

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

[100] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the Symposium on Applied Computing*, April 2006.

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

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

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

[104] 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.

[105] T. J. Klevinsky, Scott Laliberte, and Ajay Gupta. *Hack I.T.: Security Through Penetration Testing*. Addison-Wesley Professional, 2002.

[106] koders.com. Source code search engine. `http://www.koders.com`.

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

[108] Komodia, Inc. PacketCrafter. `http://www.komodia.com/tools.htm`.

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

[110] Pavel Kouznetsov. Jad. `http://www.kpdus.com/jad.html`, 2001.

[111] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access rights analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 359–372, October 2002.

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

[113] Jens Krinke. Context-sensitivity matters, but context does not. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation*, pages 29–35, September 2004.

[114] Christopher Kruegel and Giovanni Vigna. Anomaly detection of Web-based attacks. In *Proceedings of the Conference on Computer and Communications Security*, pages 251–261, October 2003.

[115] Christopher Kruegel, Giovanni Vigna, and William Robertson. A multi-model approach to the detection of web-based attacks. *Computer Networks*, 48(5):717–738, 2005.

[116] Viktor Kuncak and Martin Rinard. Boolean algebra of shape analysis constraints. In *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation*, January 2004.

[117] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the Usenix Security Symposium*, pages 177–190, August 2001.

[118] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92100, May 2004.

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

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

[121] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In *Proceedings of the International Conference on Compiler Construction*, pages 153–169, April 2003.

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

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

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

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

[126] Benjamin Livshits. The Griffin appliation security project. `http://suif.stanford.edu/~livshits/work/griffin/`, 2006.

[127] Benjamin Livshits. SecuriBench Micro. `http://suif.stanford.edu/~livshits/work/securibench-micro/`, 2006.

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

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

[130] Benjamin Livshits and Monica S. Lam. Finding security errors in Java programs with static analysis. Technical report, Stanford University, August 2005.

[131] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for Java. In *Proceedings of the Asian Symposium in Programming Languages and Systems*, pages 139–160, November 2005.

[132] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for Java. Technical report, Stanford University, November 2005.

[133] Benjamin Livshits and Thomas Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 296–305, September 2005.

[134] Benjamin Livshits and Thomas Zimmermann. Locating matching method calls by mining revision history data. In *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

[135] Johnny Long, Chris Hurley, James C. Foster, Mike Petruzzi, Noam Rathaus, Mark Wolfgang, and Max Moser. *Penetration Tester's Open Source Toolkit*. Syngress Publishing, 2005.

[136] Johnny Long, Ed Skoudis, and Alrik van Eijkelenborg. *Google Hacking for Penetration Testers*. Syngress Publishing, 2004.

[137] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: unit testing with mock objects. In *Proceedings of eXtreme Programming and Flexible Processes in Software Engineering*, May 2000.

[138] CSO Magazine and U. S. Secret Service. 2005 E-crime watch survey. `http://www.cert.org/archive/pdf/ecrimesummary05.pdf`, 2005.

[139] Michael Martin. PQL: program query language. `http://pql.sourceforge.net/`, 2006.

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

[141] Michael Martin, Benjamin Livshits, and Monica S. Lam. SecuriFly: Runtime vulnerability protection for Web applications. Technical report, Stanford University, October 2006.

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

[143] Hidehiko Masuhara and Akinori Yonezawa. Design and partial evaluation of meta-objects for a concurrent reflective language. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 418–439, July 1998.

[144] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graph construction in the presence of function pointers. Technical report, Rutgers University, 2001.

[145] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–11, July 2002.

[146] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2004.

[147] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise and efficient call graph onstruction for programs with function pointers. *Journal of Automated Software Engineering*, 2004.

[148] Nick Mitchell, Gary Sevitsky, and Harini Srinivasan. The diary of a datum: an approach to modeling runtime complexity in framework-based applications. In *Proceedings of the European Conference on Object-Oriented Programming, Systems, Languages, and Applications*, July 2006.

[149] David Moore, Vern Paxton, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer worm. *IEEE Security and Privacy*, 1(4):33–39, July 2003.

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

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

[152] Netcontinuum, Inc. Web application firewall: how NetContinuum stops the 21 classes of Web application threats. http://www.netcontinuum.com/products/whitePapers/getPDF.cfm?n=NC_WhitePaper_WebFirewall.pdf, 2004.

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

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

[155] Patrick Niemeyer. BeanShell 2.0. http://www.beanshell.org/BeanShellSlides.pdf.

[156] NIST. Vulnerability statistics generation engine. http://nvd.nist.gov/statistics.cfm.

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

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

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

[160] OWASP. Data validation (code review). http://www.owasp.org/index.php/Data_Validation_(Code_Review), June 2006.

[161] Hemant D. Pande and Barbara G. Ryder. Data-flow-based virtual function resolution. In *Proceedings of the International Symposium on Static Analysis*, pages 238–254, September 1996.

[162] Matthew M. Papi and Michael D. Ernst. Improving Java annotations to enable custom type qualifiers. http://pag.csail.mit.edu/javari/java-annotation-design.pdf, July 2006.

[163] Igor Pechtchanski and Vivek Sarkar. Dynamic optimistic interprocedural analysis: a framework and an application. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 195–210, October 2001.

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

[165] Feng Qian and Laurie Hendren. Towards dynamic interprocedural analysis in JVMs. In *Proceedings of the Usenix Virtual Machine Research and Technology Symposium*, May 2004.

[166] Derek Rayside, Steve Reuss, Erik Hedges, and Kostas Kontogiannis. The effect of call graph construction algorithms for object-oriented programs on automatic clustering. In *Proceedings of the International Workshop on Program Comprehension*, page 191, June 2000.

[167] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Bowen Alpern, Robert D. Johnson, Aaron Kershenbaum, and Larry Koved. SABER: Smart Analysis Based Error Reduction. In *Proceedings of International Symposium on Software Testing and Analysis*, July 2004.

[168] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 49–61, 1995.

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

[170] ri.gov. ri.gov security breach. `http://www.ri.gov/security/`, 2006.

[171] Erik Ruf. Partial evaluation in reflective system implementations. In *Workshop on Reflection and Metalevel Architecture*, October 1993.

[172] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[173] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 105–118, January 1999.

[174] Fred B. Schneider. Enforceable security policies. *ACM Transactions of Information and System Security*, 3(1):30–50, 2000.

[175] Marc Schönefeld. Hunting flaws in JDK. In *Proceedings of Blackhat Europe*, May 2003.

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

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

[178] Secure Software. Reflection injection. `http://www.owasp.org/index.php/Reflection_injection`, 2006.

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

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

[181] Morgan Stanley. Personal communication, June 2006.

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

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

[184] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.

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

[186] Peter Thiemann. Towards partial evaluation of full Scheme. In *Proceedings of Reflection Conference*, April 1996.

[187] James S. Tiller. *The Ethical Hack: A Framework for Business Value Penetration Testing*. Auerbach, 2003.

[188] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for Java. *ACM SIGPLAN Notices*, 34(10):292–305, 1999.

[189] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. *ACM SIGPLAN Notices*, 35(10):281–293, 2000.

[190] Oksana Tkachuk, Matthew B. Dwyer, and Corina S. Pasareanu. Automated environment generation for software model checking. In *Proceedings of the International Conference on Automated Software Engineering*, page 116, October 2003.

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

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

[193] Fredrik Valeur, Darren Mutz, and Giovanni Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, pages 123–140, July 2005.

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

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

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

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

[198] 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.

[199] Web Application Security Consortium. Web security glossary. `http://www.webappsec.org/projects/glossary/`, 2005.

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

[201] Wesley Weimer and George Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2004.

[202] John Whaley. Joeq: a virtual machine and compiler infrastructure. In *Proceedings of the Workshop on Interpreters, Virtual Machines, and Emulators*, pages 58–66, June 2002.

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

[204] John Whaley and Monica S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the International Symposium on Static Analysis*, pages 180–195, 2002 2002.

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

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

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

[208] Yichen Xie, Andy Chou, and Dawson Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Software Engineering Notes*, 28(5):327–336, 2003.

[209] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. Technical Report UILU-ENG-03-2207, University of Illinois at Urbana-Champaign, May 2003.

[210] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the Symposium on Foundations of Software Engineering*, pages 1–10, November 2002.

[211] Sean Zhang. *Practical Pointer Aliasing Analyses for C*. PhD thesis, Rutgers University, August 1998.

[212] Xiaolan Zhang, Larry Koved, Marco Pistoia, Sam Weber, Trent Jaeger, and Guillaume Marceau. The case for analysis preserving language transformation. In *Proceedings of the International Symposium on Software Testing and Analysis*, July 2006.

[213] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 145–157, June 2004.

[214] Thomas Zimmermann, Silvia Breu, Christian Lindig, and Benjamin Livshits. Mining additions of method calls in ArgoUML. In *Proceedings of the International Workshop on Mining Software Repositories Challenge*, May 2006.